CHALMERS
UNIVERSITY OF TECHNOLOGY

Path planning for autonomous buses based on Optimal Control

Master's thesis in Automatic Control & Robotics, Industrial Engineering.
(M.Sc. by Polytechnic University of Catalonia (UPC).

ALEJANDRO RODRÍGUEZ ALCÁZAR

Department of Electrical Engineering
Division of Systems and Control
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

MASTER'S THESIS 2018: NN

Path planning for autonomous buses based on Optimal Control

ALEJANDRO RODRÍGUEZ ALCÁZAR

Department of Electrical Engineering
Division of Systems and Control
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Path planning for autonomous buses based on Optimal Control

Alejandro Rodríguez Alcázar

Supervisor: Jonas Sjöberg, Department of Electrical Engineering
Examiner: Jonas Sjöberg, Department of Electrical Engineering

Master's Thesis 2018: NN
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Volvo electric bus used on line 55 in the city of Gothenburg.

Printed by Chalmers Reproservice
Gothenburg, Sweden, 2018

Path planning for autonomous buses based on Optimal Control

ALEJANDRO RODRÍGUEZ ALCÁZAR

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology

Abstract

This thesis presents an algorithm to generate trajectories for an autonomous bus approaching a bus stop. The path planning algorithm is formulated as an Optimal Control Problem (OCP) which is solved by means of nonlinear programming (NLP) using the direct multiple shooting method. This method has shown to be a good choice for solving nonlinear Boundary Value Problems (BVP) like this one -where there are constraints such as the limits of the road, the model dynamics or passengers comfort- due to its highly accurate solution and faster convergence and stability than other methods like direct single shooting methods. It uses a kinematic bicycle model with a coordinate transformation which uses the vehicle position along the path as independent variable instead of using time which permits the definition of the constraints independently of the vehicle's speed. The OCP is solved in MATLAB using CasADi, a symbolic tool for solving nonlinear optimization problems that provides high level interfaces to make the problem writing easier, in addition of having better performance than similar tools. The proposed algorithm is evaluated in multiple scenarios like different kinds of bus stops and paths inside confined areas, giving as a result a trajectory that meets with the imposed constraints successfully. Experimental tests on a real autonomous bus are carried out, resulting in a smooth bus stop manoeuvre that the passengers evaluated as fully acceptable.

.

Keywords: autonomous driving, path planning, optimal control, bus stop, Robot Operating System, Gazebo.

Acknowledgements

The work done in this thesis has been a continuation of the work started by Mario Zanon, researcher at IMT School for Advanced Studies Lucca (Italy). I am really thankful to him for explaining me how the algorithm works and helping me with several doubts I had along the project.

I would also like to thank to my tutor Jonas Sjöberg, who has helped, guided and provided valuable insights through different questions on the development of this thesis.

Thanks to Nandish, Jafar and Joakim, for showing me so many interesting things in Volvo, for having collaborated in this project and helping me to be able to test in the simulator and the real bus.

A special acknowledgment to all friends that I have met during my stay in Sweden, who made me feel as if I was at home, sharing and living many wonderful moments and being a great support to me when needed.

Finally, I would like to give my most special thanks to Musa, who is not here anymore, but was the biggest support I had along my years at university with her happiness each time I went back home.

Alejandro Rodríguez Alcázar, Gothenburg, 2018.

Contents

List of Figures

List of Tables

1

Introduction

There is without a doubt an increment in the development of strategies focused on enhancing the socio-economic, logistic, competitive and ecological performance of cities [1]. From the means of transport point of view, it could be said that any environment which is classified as a smart city should respond consequently to the changes in mobility, increase of urbanisation and traffic congestion. Therefore, improving the public transport network will result in a more sustainable, efficient and environmental friendly transport system. Many projects are being studied and implemented nowadays by main cities to improve this public transport, like the use of electrical buses to reduce the levels of noise, CO_2 or NO_x [2] [3], but also more ambitious projects like the use of electric autonomous vehicles, like public transport buses, that will lead into a safer, more efficient and comfortable mobility [4]. Implementing a network of autonomous buses in a city involves many areas of study, as they are a combination of mechanics, robotics, IT and electronics and they should be able to drive accordingly to the traffic circumstances and surroundings safely and comfortably. For that, it is of major importance the developing of algorithms like the one presented in this thesis for generating paths which will allow autonomous buses to reach their destinations satisfying the inhabitants needs.

1.1 Problem description

The problem of finding a path for a bus that goes from a starting point with an initial configuration to an ending point is considered in this thesis. For instance, the scenario of an autonomous bus that is approaching a bus stop can be considered:

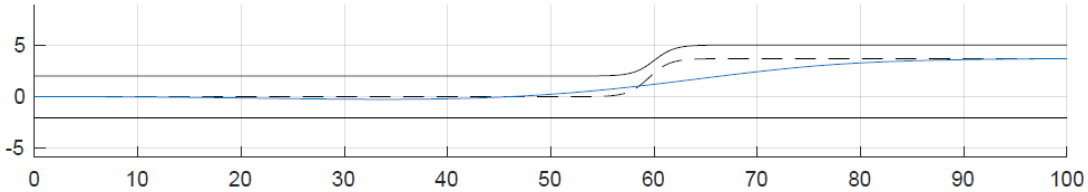


Figure 1.1: Initial scenario description for a left side bus stop. Black solid lines delimit the lane and bus stop geometries. Blue line shows an optimal path and black dashed line is a reference path

The scenario described in Figure 1.1 shows a lane with a left side bus stop that starts approximately in the middle of the lane length. The bus comes with an initial speed and it has to stop at a certain final distance from the origin. The dashed line shows a reference trajectory and the blue line an example of a trajectory that the bus can follow to perform a bus stop. The problem is then finding a trajectory that drives the bus until the stop without going outside of the lane and also meeting with a series of constraints like velocity and final position constraints or more related to comfort aspects like jerk or lateral acceleration constraints. There exist different approaches for solving this kind of problems, being the formulation of it as an Optimal Control Problem (OCP) and its solution one of them and the one used in this thesis.

1.2 Scope

This master's thesis is part of a larger project funded by Fordonsstrategisk Forskning och Innovation (FFI) that involves the driving of autonomous city buses. This project will run for 3 years and is focusing on three main parts which are an autonomous bus stop docking and take-off, an autonomous depot deployment and a several buses platform communicating between them as it can be seen in Figure 1.2. This thesis proposes a global path planner algorithm to generate a trajectory that an autonomous bus should follow in order to have a complete bus stop docking. This trajectory has to meet certain requirements regarding safety and comfort of the passengers, adding this requirement as constraints for the problem. The algorithm can be used in many different bus stop geometries and other modifications to allow the bus to take-off and drive in a controlled track are also performed along with a formulation of the problem that minimizes the time used in following the trajectory. Finally, another purpose of this thesis is to provide help in implementing a virtual simulator where the trajectories can be evaluated. This simulator includes a model of the bus and a path follower. This path follower has the mission of processing the information from the global planner trajectory in order to give the right orders to the actuators in the bus to follow it. This path follower is not in the scope of the thesis, although some help has been given for the development of the simulator that will be used. This tool will be helpful in later states of the project, as it will allow to test different algorithms and get results to see if the requirements which are needed are met.

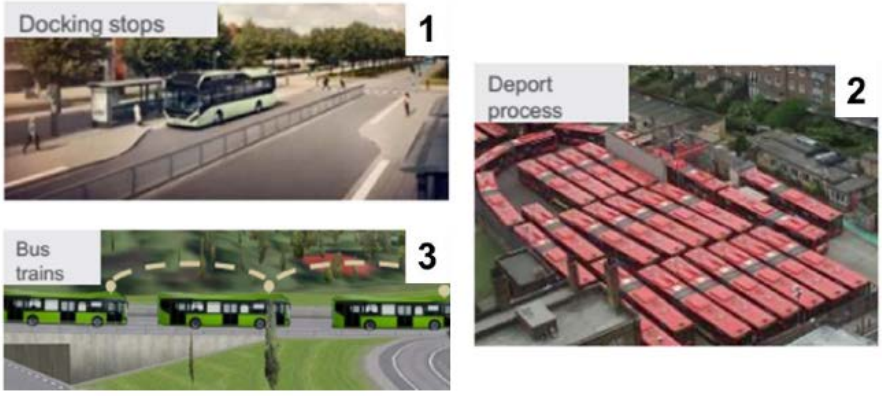


Figure 1.2: Scope of the FFI project with three possible scenarios. Bus stop dockings (1), depot deployment (2) and interconnected buses (3). [5].

1.3 Contributions

An algorithm which generates a trajectory for the bus between two points is implemented. This trajectory meets certain comfort variables limits in order to have a proper autonomous driving that allows the passengers to have a comfortable trip, such as maximum and minimum speed, limits on the longitudinal jerk, lateral acceleration, steering rate but also, the trajectory has to guarantee the safety of the passengers by not going over the sidewalk or hitting the limits of the lane. These constraints are formulated mathematically and introduced into an optimization problem whose solution leads to the desired trajectory.

Having as a starting point the solution for the trajectory generation for a left bus stop docking as an OCP, new capabilities have been added to the original problem, such as setting it for different geometries of bus stops, modify the constraints of the problem to have a bus take-off, a time optimal formulation or to be able to drive around a controlled area. The result is an algorithm capable of finding an optimal trajectory for that kind of situations.

In addition to the previous, this trajectory can now be used in a bus simulator and in a prototype of an autonomous bus. Before, the trajectories had to be recorded beforehand by driving manually and then they were replicated and now, by exporting the trajectory generated in this thesis, computer-generated trajectories can be used. Finally, these computer-generated trajectories have been tested in a real bus in a test track resulting in a smoother driving than when following the manual driving recorded trajectories.

1.4 Thesis outline

The rest of this thesis is structured as follows:

- Chapter 2 gives a scope of the problem that has to be solved, how is it implemented as an OCP, and explains briefly different optimization methods like single direct shooting or the used multiple direct shooting algorithms.

- Chapter 3 focuses on the solution of the OCP and shows an introduction to the software used for its solution. Then it is explained some modifications that have been made to the algorithm. Finally, it explains the architecture of the simulator based on ROS and Gazebo that is used for testing the trajectory with a dynamic model of the bus.

- Chapter 4 presents the results of the OCP showing the trajectories that have been obtained for different scenarios and the results obtained from the simulation in Gazebo and the real test.

- Chapter 5 closes this thesis with the obtained conclusions and future improvements that can be done in this project.

2

Problem definition and relevant theory

In this chapter the problem of path planning for an autonomous bus that should meet some requirements such as comfort variables or boundary limits will be explained. It will also show how the problem is formulated as an Optimal Control Problem (OCP), and a small study of different methods to solve OCPs will be carried out, choosing at the end the one most suitable method to solve it.

2.1 Definition of the problem

In order to define the problem, the case of an autonomous bus approaching a bus stop is considered as mentioned in 1.1. More in advance it is shown how the constraints of the problem can be modified to solve other situations like a take-off, a circular trajectory or how to implement different kinds of bus stops.

.

As mentioned in section 1.2, several constraints appear in this problem. One is that the bus cannot go outside the boundaries of the lane or hit the borders. Other constraints are related to the ending point where the bus has to stop, maximum and minimum values in lateral acceleration, jerk or velocity. These constraints can be modified to obtain different lane geometries, or allow the bus to reach higher speeds, turning faster or accelerating and braking harder.

The solutions that have been obtained in this thesis are presented as a global trajectory in a high level that connects a starting point with an ending point. According to [6] and [7], the usage of a bicycle model with perfect rolling constraints like the one shown in Figure 2.2 for modelling the bus is good enough for control purposes. In other papers where there is also a high-level global path planner and a low-level path follower, a kinematic bicycle model is used for the path planner and a more accurate model (a four-wheel vehicle dynamical model) is used for the path follower [8]. Several reasons are behind the choice of a simple bicycle model being the faster computation time and the simplicity of the model the main reasons which are important in order to have this algorithm working in real time in future works. Due to the fact of working with a bus, low speeds, and high masses are expected, allowing the neglection of road-tire

interaction, having a slip-free bicycle model. However, for the path follower purposes, more complex models are used. The simple time dependent bicycle model with perfect rolling constraints as the one shown in Figure 2.2 is thus governed by the following equations:

$$\dot{\psi} = \frac{v_x}{L}\tan(\delta) \tag{2.1}$$

$$\dot{X} = v_x \cos(\psi) \tag{2.2}$$

$$\dot{Y} = v_x \sin(\psi) \tag{2.3}$$

where the states are $\xi_t = (X, Y, v_x, a, \psi, \delta)$ being X, Y the coordinates from the midpoint of the rear axle of the bus in global coordinates, the longitudinal velocity, acceleration, orientation and steering angle of the bus respectively. L denotes the wheelbase of the bus. For comfort aspects, the variation of the acceleration known as jerk (represented as b) and the steering rate (ω) will be the controls chosen for this problem: $u_t = (b, \omega)$.

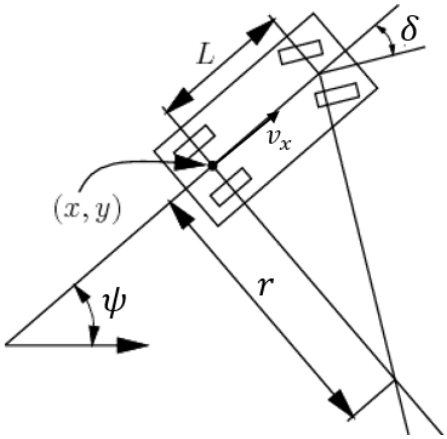


Figure 2.1: Bicycle vehicle kinematic model.

The system can be stated as follows:

$$\dot{\xi}_t = f_t(\xi_t, u) = \begin{bmatrix} v_x \cos(\psi) \\ v_x \sin(\psi) \\ a \\ b \\ \frac{v_x}{L}\tan(\delta) \\ \omega \end{bmatrix} \tag{2.4}$$

This model dynamics are time-dependent and a transformation to spatial (track) dependent dynamics is proposed [8] [9]. This reformulation is done to avoid the assumption of using a constant longitudinal speed that is made in other works. Thus, by passing from a time-dependent vehicle dynamic to a position-dependent dynamic, obstacle constraints can be defined independently of the vehicle speed, having a more natural formulation of the obstacles and general road bounds which means an easier

optimal control formulation and also allows to have time as a state variable which can be optimized.

The coordinates of the vehicle over the global frame are denoted as $[X, Y]^T$, which will be projected on the centreline of the track parametrized by its arc length and denoted as σ(s).

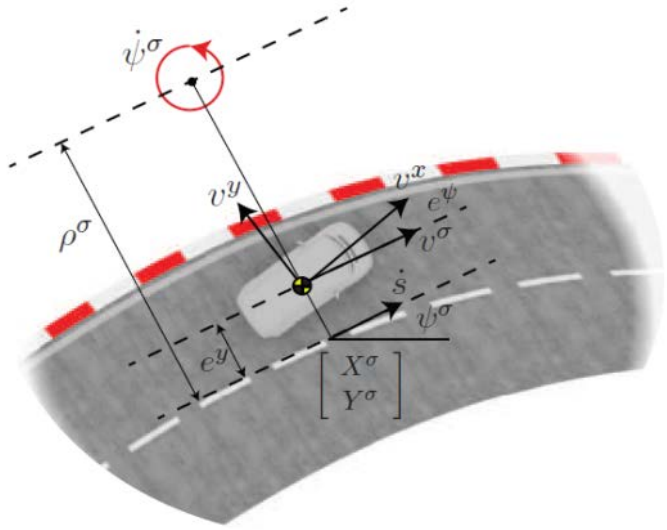


Figure 2.2: Coordinate system used in the spatial transformation of the vehicle dynamics. The s coordinate represents the arc length along the track [9].

According to Figure 2.3, $[X^\sigma, Y^\sigma]^T$ and ψ^σ are the position and orientation of the reference point on the path $\sigma(s)$. The states X, Y and ψ can be replaced by the lateral displacement e_y and the angular deviation e_ψ respect to the centreline of the path. Longitudinal velocity is denoted by v_x and $\dot{\psi}^\sigma$ is the vehicle's yaw rate. The time derivative will be related to the spatial derivative by:

$$\xi' = \frac{d\xi}{ds} = \frac{d\xi}{dt}\frac{dt}{ds} = \frac{\dot{\xi}}{\dot{s}} \tag{2.5}$$

where $\dot{s} = \frac{ds}{dt}$ is the vehicle's velocity along the path and it is represented by:

$$\dot{s} = \frac{1}{1 - \kappa_\sigma \cdot e_y} \cdot v_x \, cos(e_\psi) \tag{2.6}$$

being κ_σ the local curvature of the centreline of the curve σ. Hence the new introduced states are shown below, denoting the state derivatives as $\xi'(s)$ instead of $\dot{\xi}(t)$:

$$\xi'(s) = \left[e_y', v', a', e_\psi', \delta', t'\right] \tag{2.7}$$

$$e_y'(s) = \frac{v_x \sin(e_\psi)}{\dot{s}}; \quad v'(s) = \frac{a}{\dot{s}}; \quad a'(s) = \frac{b}{\dot{s}}; \quad e_\psi'(s) = \frac{\dot{\psi}}{\dot{s}} - \kappa_\sigma; \quad \delta'(s) = \frac{\dot{\delta}}{\dot{s}} \tag{2.8}$$

Time information can be recovered by solving the differential equation: $t'(s) = 1/\dot{s}$. One important thing that has to be taking into account is the fact of having a division by $\dot{s}$ to have the spatial formulation, which introduces a singularity when v tends to 0. Therefore, special care has to be taken when reaching low velocities in order to avoid that singularity.

In case the global coordinates have to be recovered, it can be done using the following transformation from the spatial coordinates to the global coordinates:

$$\begin{aligned} X &= X^{\sigma} - e_y \sin(\psi^{\sigma}) \\ Y &= Y^{\sigma} + e_y \cos(\psi^{\sigma}) \\ \psi &= \psi^{\sigma} + e^{\psi} \end{aligned} \tag{2.9}$$

In next section it will be seen how these spatial coordinates transformation is used when formulating the Optimal Control Problem.

2.2 Optimal Control Problem formulation

Using the spatial dynamics formulation allows to model obstacles and lane boundary constraints as simple bounds on the state vector. If a variable speed of the bus is considered, this formulation avoids the non-convex speed dependent (and implicitly input dependent) constraints from the time formulation in the global coordinate system [9].

Denoting the vectors of the states and control inputs by ξ and u respectively, the control problem is formulated as an Optimal Control Problem (OCP) over the horizon defined by the interval $[S_o, S_f]$ in space, formulated as [9]:

$$\min_{\mathrm{x,u}} \int_{S_0}^{S_f} L(\xi, u)\, ds + M\left(\xi\left(S_f\right)\right) \tag{2.10}$$

$$\begin{aligned} s.t. \quad & \xi(S_0) = \xi_0 \\ & \xi'(s) = f(\xi, u) && s \in [S_o, S_f] \\ & h(\xi, u) \leq 0 && s \in [S_o, S_f] \\ & g(\xi(S_f)) = 0 \end{aligned}$$

where the cost is defined as the integration of a running cost L and a final cost M, formulated as a least-squares objective function which aims at tracking a state reference trajectory, denoted by ξ_{ref} and u_{ref}, while taking a control term into account:

$$L(\xi, u) = \left(\xi - \xi_{ref}\right)^T Q \left(\xi - \xi_{ref}\right) + \left(u - u_{ref}\right)^T R \left(u - u_{ref}\right) \tag{2.11}$$

$$M\left(\xi\left(S_f\right)\right) = \left(\xi\left(S_f\right) - \xi_{ref}\left(S_f\right)\right)^T P \left(\xi\left(S_f\right) - \xi_{ref}\left(S_f\right)\right) \tag{2.12}$$

The functions $h(\xi,u)$ and $g(\xi(S_f))$ define the path and terminal constraints respectively. Note that for the case of a bus stop, the constraints required at the end are equal to 0. The function $f(\xi,u)$ defines the ODE model with the equations shown above, the function $h(\xi,u)$ is defined such that $a_y \epsilon [a_{y,min}, a_{y,max}]$, $b \epsilon [b_{min}, b_{max}]$, $v \epsilon [v_{min}, v_{max}]$. The function $g\left(\xi(S_f)\right)$ is defined such that $g\left(\xi(S_f)\right) = (e_y - e_{y,end}, e_\psi, v - v_{min}, a, \delta)$. This final configuration will bring the bus to a low but non-zero velocity due to the singularity mentioned above. The final part of the trajectory can be easily computed in time domain e.g. from 1 km/h to 0 km/h to have a full bus stop. The matrices Q and R, are weighting matrices for the states and the control signals respectively, having Q a size of 5x5 and R a size of 2x2. Matrix P is a terminal weighting matrix that is simply implemented with the same values as matrix Q.

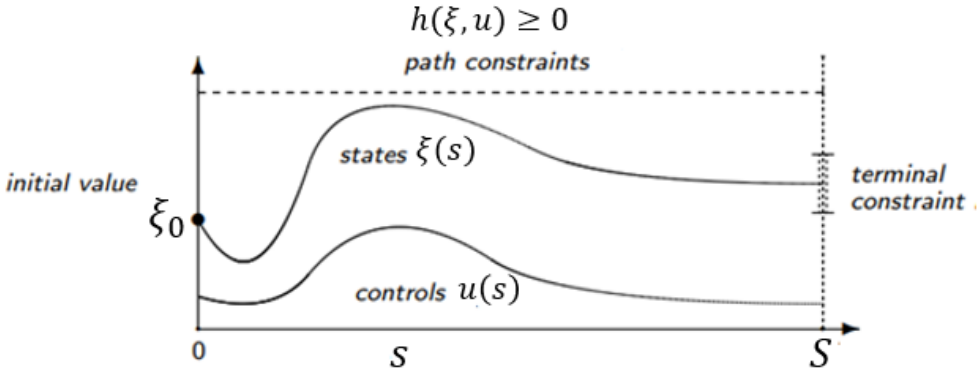


Figure 2.3: Representation of the simplified Optimal Control Problem in ODE in space domain [10].

In addition to these constraints, side collision avoidance is implemented so the two sides of the bus do not go outside the lane boundaries nor go above the sidewalk. If the left and right side of the lane geometry are denoted by e_y^l and e_y^r respectively, then the equations that define this constraint are:

$$e_y + l\sin(\psi) + \frac{w}{2}\cos(\psi) \leq e_y^l \qquad l \epsilon \left[-L_r, L + L_f\right] \tag{2.13}$$

$$e_y + l\sin(\psi) - \frac{w}{2}\cos(\psi) \leq e_y^r \qquad l \epsilon \left[-L_r, L + L_f\right] \tag{2.14}$$

where l is a coordinate spanning the longitudinal direction of the vehicle, w is the width of the bus, L_r and L_f denote the distance from the rear/front part of the bus from the rear and front axles respectively.

2.3 Optimal Control Problem solution.

Several methods exist to solve the Optimal Control Problems like the simplified OCP in ordinary differential equations (ODE) shown in the previous section, such as dynamic programming, indirect methods, or direct methods such as the single-shooting, collocation or multiple-shooting algorithms [10].

The dynamic programming approach [11] uses the principle of optimality of sub arcs to compute recursively a feedback control for all times and initial states. Different methods to numerically compute the solution exist but they are restricted to small state dimensions.

Indirect methods work as "first optimize, then discretize". They use the necessary conditions of optimality of the infinite problem to derive a boundary value problem (BVP) in ODE. However, it has some drawbacks such as the difficulty in solving the differential equations due to nonlinearity and instability and changes in control structure usually require a new problem setup [10].

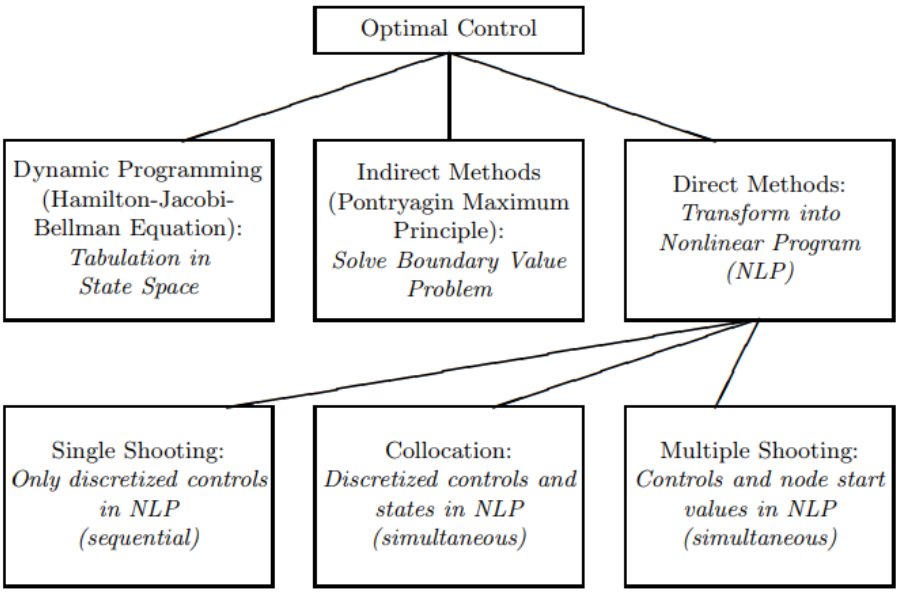


Figure 2.4. Overview of numerical methods for solving OCPs [10].

The direct methods work in the way of "first discretize, then optimize" and transcribe an infinite problem into finite dimensional Nonlinear Programming Problem (NLP) which has to be solved. Nowadays they are the most commonly used to solve BVP due to their easy applicability, robustness and the easier treatment of the inequality constraints compared to the indirect methods. As shown in Figure 2.5., inside direct methods there are different approaches: sequential approaches (direct shooting) and simultaneous approaches (collocation and multiple shooting algorithms). All of them discretise the control trajectory but they differ in how the state trajectory is handled.

In the sequential approaches, the state trajectory is taken as an implicit function of controls so the simulation and optimization iterations proceed sequentially and the NLP has only the discretized control as optimization degrees of freedom. In direct single shooting the control is parametrized using a piecewise constant on the finite dimensional grid and using an explicit expression for the controls, the whole state trajectory can be eliminated from the OCP, having to solve an NLP only in the discretized controls. It has some advantages like that only the initial guess for the control is needed but it has disadvantages such as the knowledge of the state trajectory in the initialization cannot be used, or unstable systems are difficult to treat.

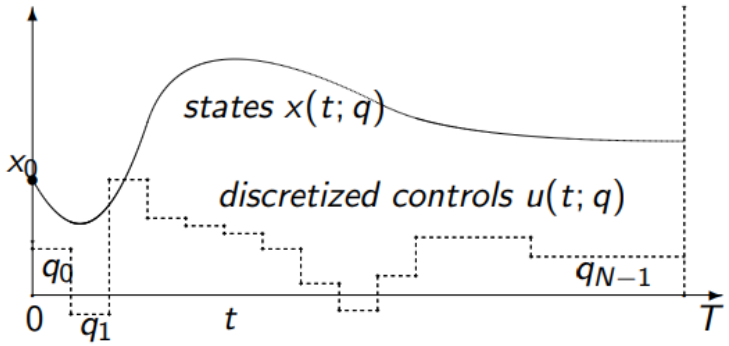


Figure 2.5: Overview of the single shooting method [10].

However, the OCP in this thesis is solved by a simultaneous approach, concretely the Bock's multiple shooting method [12]. This method also starts solving the OCP by discretizing the control signal as a piecewise constant on a coarse grid:

$$u(s) = u_k \qquad s \in [s_k, s_{k+1}] \tag{2.15}$$

but the difference with the single shooting approach is that the ODE is solved on each interval $[s_k, s_{k+1}]$ starting with an artificial initial value μ_k for each interval and it keeps the initial states of all the shooting intervals as optimization variables, while the single shooting eliminates all the states by a forward simulation:

$$\dot{\xi}_k(s) = f(\xi_k(s), u_k) \tag{2.16}$$

$$\xi_k(s_k) = \mu_k \tag{2.17}$$

obtaining trajectory pieces $\xi_k(s; \mu_k, u_k)$. Simultaneously with the ODE solution, the following integrals are solved:

$$\ell(\xi, u) = \int_{s_k}^{s_{k+1}} L(\xi_k(s_k; \mu_k, u_k), u_k)\, ds \tag{2.18}$$

and to constraint the artificial degrees of freedom μ_k, the following equality constraint to ensure continuity is imposed:

$$\mu_{k+1} = \xi_k(s_{k+1}; \mu_k, q_k) \tag{2.19}$$

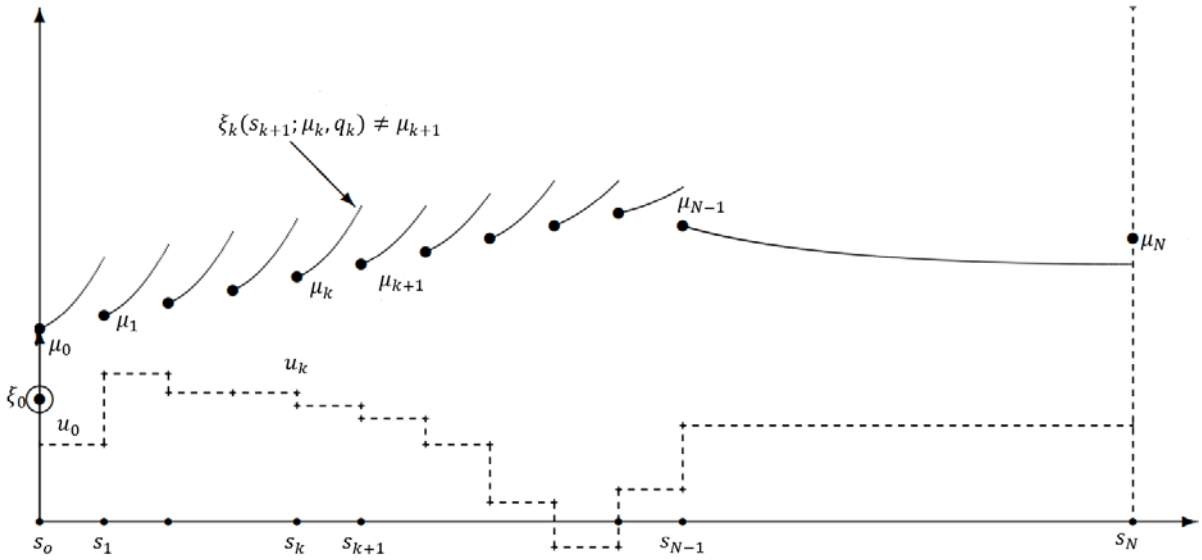


Figure 2.6: Overview of the multiple shooting method [10].

Then the OCP can be read then as the following NLP formulation:

$$\min_{x,u} \sum_{k=0}^{N} \ell(\xi, u) + M\left(\xi(S_f)\right) \tag{2.20}$$

$$\begin{aligned} s.t. \quad & \xi(S_0) = \xi_0 \\ & \xi_{k+1} = f^d(\xi_k, u_k) && k = 0, \dots, N \\ & h(\xi_k, u_k) \leq 0 && k = 0, \dots, N; \;\; l = 0, \dots, N_l \\ & g(\xi_{N+1}) = 0 \end{aligned}$$

with:

$$\ell(\xi, u) = \left(\xi - \xi_{ref}\right)^T Q \left(\xi - \xi_{ref}\right) + \left(u - u_{ref}\right)^T R \left(u - u_{ref}\right) \tag{2.21}$$

$$M\left(\xi(S_f)\right) = \left(\xi(S_f) - \xi_{ref}(S_f)\right)^T P \left(\xi(S_f) - \xi_{ref}(S_f)\right) \tag{2.22}$$

State and control constraints are relaxed to the finite time grid. The collision avoidance constraint is also relaxed and enforced only on a finite grid of N_l points that discretize the sides of the bus. The benefits of simultaneous methods and in concrete direct multiple shooting is the allowance to use ODE/DAE solvers; that the knowledge of the state trajectory in the initialization can be used; it also shows fast local convergence, it can treat unstable systems well and deals with state and terminal constraints.

The resulting OCP is solved in MATLAB using an open source software tool called CasADi for numerical optimization in optimal control involving ordinary differential equations or differential algebraic equations including NLP solvers and the use of symbolic expressions [13].

3

Algorithm implementation

3.1 The symbolic tool for numerical optimization CasADi

CasADi is defined as symbolic tool for algorithmic differentiation and gradient based numerical optimization with strong focus on optimal control. One of the main advantages that it has is the use of the syntax of Computer Algebra Systems (CAS), allowing the user to construct symbolic expressions that can be differentiated in an efficient way using different algorithms for algorithm differentiation [14].

The main purpose of CasADi is to give a low-level interface to the user for quick and high efficient implementation of algorithms for nonlinear numerical optimization and formulate nonlinear programs problems (NLP) or optimal control problems (OCP) as the one mentioned in the previous section. Furthermore, it is available for Linux, OS X and Windows and can be used in Python, C++ and MATLAB environments.

The reason to use CasADi is that, compared to other symbolic toolboxes that deals with symbolic expressions such as sympy, yalmip or mupad, CasADi is much faster even for a large set of variables as it can be seen in the Figure 3.1. This, together with the flexibility of the programming language that it offers, makes CasADi to be a good choice for this problem [15].

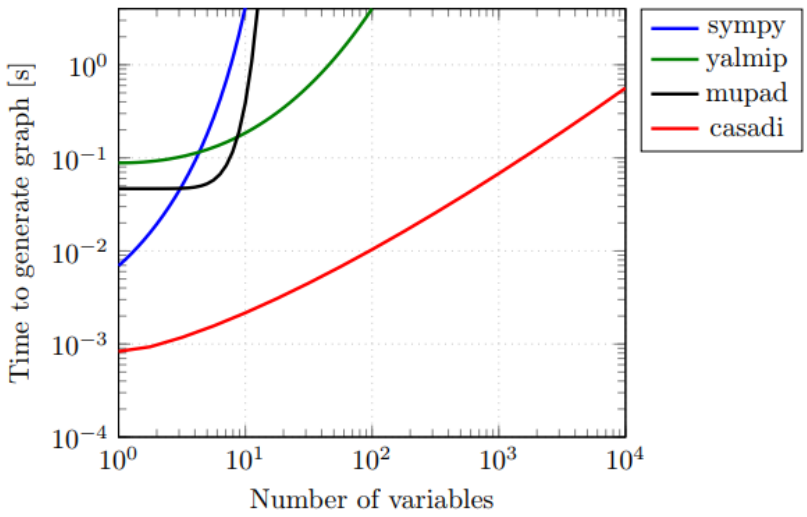


Figure 3.1: Comparison of different symbolic tools [15].

CasADi reformulates OCP problems into NLP problems. It implements several NLP solvers to solve non-linear problems of the form:

$$\begin{aligned} &\min_{x} f(x,p) \\ s.t.\quad &x_{lb} \le x \le x_{ub} \\ &g_{lb} \le g(x,p) \le g_{ub} \end{aligned} \tag{3.1}$$

where x is the decision variable and p is a parameter vector. An NLP solver for CasADi is a vector that takes p, the bounds $(x_{lb}, x_{ub}, g_{lb}, g_{ub})$ and an initial guess for the solution and returns the optimal solution. The most popular NLP solver used in CasADi is IPOPT [16] and interior point solver that can solve very large NLPs and has a large and active user community, but other NLP solvers such as SNOPT, WORHP and KNITRO can be interfaced with CasADi automatically. For example, for the following problem:

$$\begin{aligned} \min_{x,y,z} \quad & x^2 + 100\,z^2 \\ s.t. \quad & z + (1-x)^2 - y = 0 \end{aligned} \tag{3.2}$$

an NLP solver using IPOPT can be written in CasADi in MATLAB as follows:

```
% Define symbolic variables
x=SX.sym('x'); y=SX.sym('y'); z=SX.sym('z');
% Create nlp solver
nlp=struct('x', [x;y;z], 'f', x^2+100*z^2, 'g', z+(1-x)^2-y)
% Solve
S=nlpsol('S', 'ipopt', nlp)
```

CasADi solves OCPs by using different methods such the ones mentioned before like indirect (optimize then discretize), or direct (discretize then optimize) methods like single or multiple shooting by reformulating the OCP into NLP problems. The user has to write his own OCP solver by defining correctly the constraints, parameters and optimization variables, but CasADi helps with this by providing several high-level function blocks, making it easier to write the problem with the syntax needed by the NLP solver. It includes several functions, that work with symbolic variables which make the coding of the problem easier, apart from the typical arithmetic operations or trigonometric or exponential functions, like logical operators, conditional functions such as if-else, creation of 1D or 2D lookup tables or the capability of writing your own symbolic functions with the required inputs and outputs [14].

Extra functionalities such as C code generation is extremely useful. CasADi can autogenerate C code for a large subset of function objects. This has multiple benefits like the speeding up of the evaluation time of the functions; the allowance of using CasADi in machines that do not have it installed (only a C compiler will be needed) and the autogenerated code can be debugged to detect, for instance, slow processes that take place during the evaluation [14].

3.2 Writing the Optimal Control Problem in MATLAB with CasADi

In this section it will be explained some important parts related to the codes that have been used to solve the different problems exposed in this thesis. It will clarify how the implementation of the Optimal Control Problems (OCPs) in MATLAB and CasADi is done.

Considering the initial problem of a lane change bus stop to the left, shown in Figure 3.2, where the bus comes with an initial speed of 45 km/h and has to stop at a distance $S_f = 100m$ at the end of the bus stop. The geometry of the upper part of the lane (including the bus stop) is defined by the function:

$$yS(x) = \frac{3}{1 + e^{-a\cdot(x - x0S)}} \tag{3.3}$$

that approximately gives the shape of a bus stop with 3 m width. $x0s$ defines the beginning of the bus stop and a allows to modify the slope of the beginning of the bus stop. Note that this geometry does not represent a real bus stop and is used for explaining purposes. According to the direct multiple shooting method explained in previous sections, this space has to be divided into N steps of a grid, having a sample space $ss = Sf/N$.

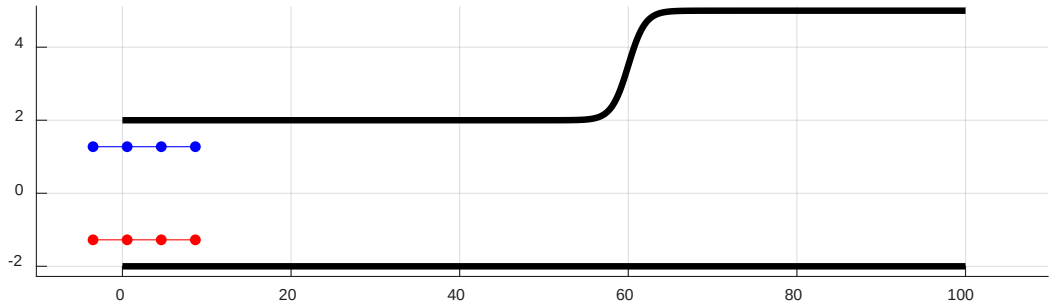


Figure 3.2: Left bus stop with the bus in its initial position. Both sides of the bus are discretized in 4 points. Left side of the bus is shown in blue and right side in red.

The bus dimensions are taken from the datasheet of the Volvo 7900 Electric Hybrid Bus (see appendix A). As defined in equations (2.13) and (2.14), the geometric parameters are $L = 5.945\ m, L_r = 3.485\ m, L_f = 2.704\ m$ and a width $w = 2.55\ m$.

The sides are discretized in N_l points and the system dynamics from equations (2.4) are simulated using a Runge-Kutta integrator of order 4 (RK4) with a number of integration steps defined by the variable n_int_steps:

$$\xi_{i+1} = \xi_i + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.4}$$

with:

$$\begin{cases} k_1 = f(x_i, \xi_i) \\ k_2 = f\left(x_i + \frac{1}{2}h, \xi_i + \frac{1}{2}k_1 h\right) \\ k_3 = f\left(x_i + \frac{1}{2}h, \xi_i + \frac{1}{2}k_2 h\right) \\ k_4 = f(x_i + h, \xi_i + k_3 h) \end{cases}$$

where h is the size of the interval: $h = ss/n\_int\_steps$. This RK4 formula is implemented as a symbolic function in CasADi. Then, for each point k of the interval $1{:}N$ with sample space ss, containing the points of the grid, the problem is first constructed symbolically by computing the symbolic RK4 and setting the corresponding vector of constraints for each point j of the N_l points that discretize the side of the bus. To set the constraints that define the upper and lower bounds of the geometry of the road, firstly, the symbolic position (x_p, y_p) of each point of the side of the bus is computed using a rotation matrix:

$$x_p = N_{l_j} \cos(\psi_p) - \frac{w}{2}\sin(\psi_p) \tag{3.5}$$

$$dy_p = N_{l_j} \sin(\psi_p) \pm \frac{w}{2}\cos(\psi_p) \tag{3.6}$$

$$y_p = e_{y_k} + dy_p \tag{3.7}$$

then, the upper bound at instant k is defined by the function that describes the upper part of the lane geometry (3.3) evaluated at the point $x = (k-1)\cdot ss + x_p$:

$$e_{y(\max),\mathrm{k}} = \frac{3}{1 + e^{-a\left((k-1)ss + x_p - x0S\right)}} \tag{3.8}$$

Note that the term $(k-1)\cdot ss$ gives the coordinate x in meters along the lane and the adding term x_p corresponds to the position of each one of the N_l points of the bus. Then, the constraint that makes that none of the points of the bus side go further than $e_{y(max),k}$ is defined by:

$$g_u = \left(y_p - e_{y(\max),\mathrm{k}}\right) \tag{3.9}$$

The same has to be done for the lower bound/right part of the lane. For this scenario, the lower bound $e_{y(\min),\mathrm{k}} = -2\,m$ for all the N points of the grid and therefore $g_l = \left(y_p - e_{y(\min),\mathrm{k}}\right)$. (Note, that in equation (3.6), $\pm$ is defined for the left and right sides of the bus respectively.) An array denoted as g containing g_u and g_l will be passed to the CasADi NLP solver along with the vector of states V, reference V_r and the cost function implemented according to the OCP definition in (2.20). Proceeding in the same way as in the example of how to create an NLP solver in section 3.1, the OCP is now transformed into a Nonlinear Program using direct multiple shooting method handed by the CasADi's function *nlpsol* that generates the necessary derivatives using algorithmic differentiation and pass the OCP to the NLP solver IPOPT as follows:

```
nlp = struct('x', V, 'p', Vr, 'f', cost, 'g', g);
solver = nlpsol('solver', 'ipopt', nlp);
```

The geometry constraints are not the only ones that shall be implemented in the problem. As it was mentioned at the beginning of the thesis, it is a must that the solution meets other requirements like maximum and minimum lateral acceleration, longitudinal acceleration, jerk, and speed. This constraints for the states and the controls are defined in arrays of length N+1 and N for the states and the controls respectively. For instance, to set the upper ($ubv = 1\ m/s^2$) and lower ($lbv = -1 m/s^2$) bounds for the acceleration along the grid of N points it will be defined as:

```
lbv(iV('x',':','a')) = -axmax*ones(size(iV('x',':','a')));
ubv(iV('x',':','a')) =  axmax*ones(size(iV('x',':','a')));
```

With CasADi's *nlpsol* function, only inequality constraints can be used to formulate NLPs, so in case an equality constraint is needed (i.e. Final position of the centre of the rear axle along the Y axis), the inequality constraints have to be reformulated as:

```
lbv(iV('x',N+1,'e_y'))= ey_stop;
ubv(iV('x',N+1,'e_y'))= ey_stop;
```

which, given the values of the needed variables for this case, the expressions above can be read as:

$$e_{y\ stop} \leq e_{y,k=N+1} \leq e_{y\ stop} \tag{3.10}$$

Finally, when the problem has been completely defined, the *solver* that has been created before is called, passing to it the initial conditions of the states, the vectors of constraints mentioned above and a vector containing the reference for the states. CasADi will solve the OCP returning a vector containing the optimal solution for the states $\xi'(s) = \left[e'_y, v', a', e'_\psi, \delta', t'\right]$ and controls $u(s) = [b, \omega]$ along the defined grid of N points.

3.3 Constraints and cost function modifications

Before it was shown the case of a bus that was approaching a bus stop to the left. The constraints explained in the previous section and the initial condition of the states can be changed easily to have other scenarios of interest like different bus stops or having a bus take-off. Next sub-sections explain and show examples of how the methodology followed in 3.2 to write the case of a bus stop as an OCP in MATLAB can be modified to solve other problems.

3.3.1 Different bus stop geometries

In cities, different kinds of bus stops can be found. The Swedish Road Administration (Vägverket), published a document providing information about how bus stops should be designed in urban and rural environments [17] taking into account several factors like the traffic, crossings, cycle paths or passenger needs. Different styles of bus stops are shown

in Figure 3.3. Therefore, it is interesting to have an algorithm that works for different styles of bus stops.

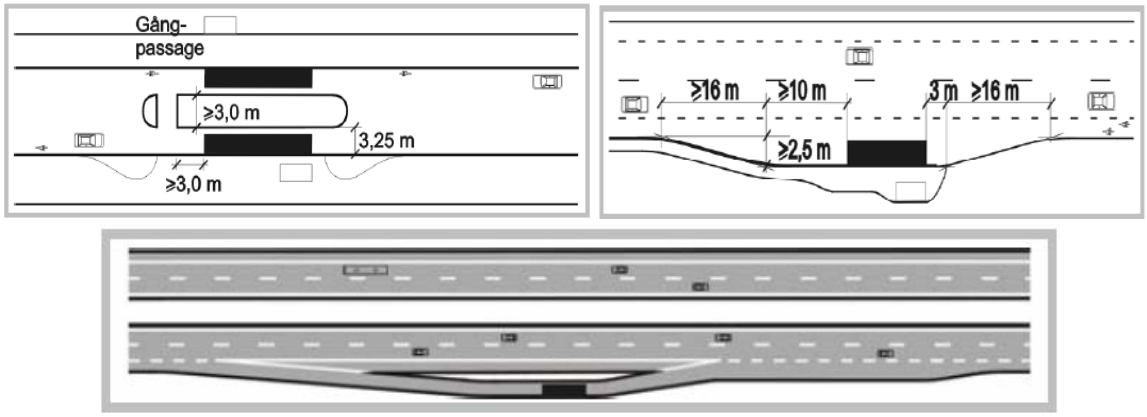


Figure 3.3: Different bus stops geometries. Straight ahead bus stop (top left), single lane change (top right), extra outside lane (bottom). Black rectangle represents the bus [17].

. Previously, it was shown that the way of setting the boundaries of the lane, was done by modifying the constraints $e_{y(\max)}$ and $e_{y(\min)}$ that defined the maximum and minimum values that each of the N_l points that discretise the bus sides can reach at each instant k from the grid of N points. Hence, it is easy to see that by using different expressions than (3.3), will lead into different boundaries for different geometries of bus stops dockings. For instance, by changing the values of $a, x0S$ and $stopwidth$, in equation (3.3), the values of $e_{y(\max)}$ and $e_{y(\min)}$ can be changed and therefore different geometries of bus stops like the ones seen in [17] can be approximated:

- Two lanes that converge into one lane where the bus stop is. (A positive value of a has been used to modify the upper bound of the geometry, having now a sidewall profile that goes down).

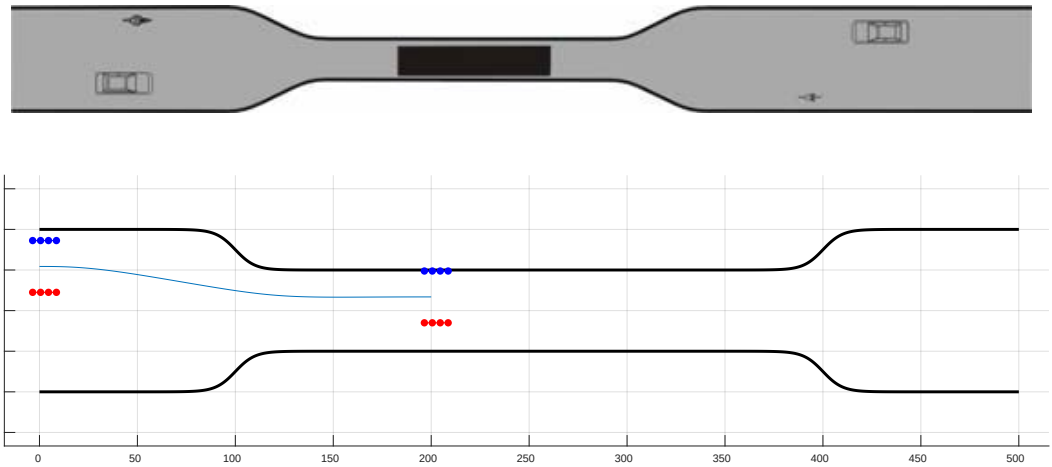


Figure 3.4: Representation of a bus stop in MATLAB where two lanes converge into one lane. Starting and ending points of the bus are shown. Left bus side (blue), right bus side (red) [17].

- Simple stop where vehicles in the same direction cannot pass while the bus is stopped. (In this case the upper bound is set continuously to be 3m and the lower bound has been set using the equation (3.3) so the low boundary line starts in the centre of the lane at -1.5m and ends at 0m according to the obstacle in the centre).

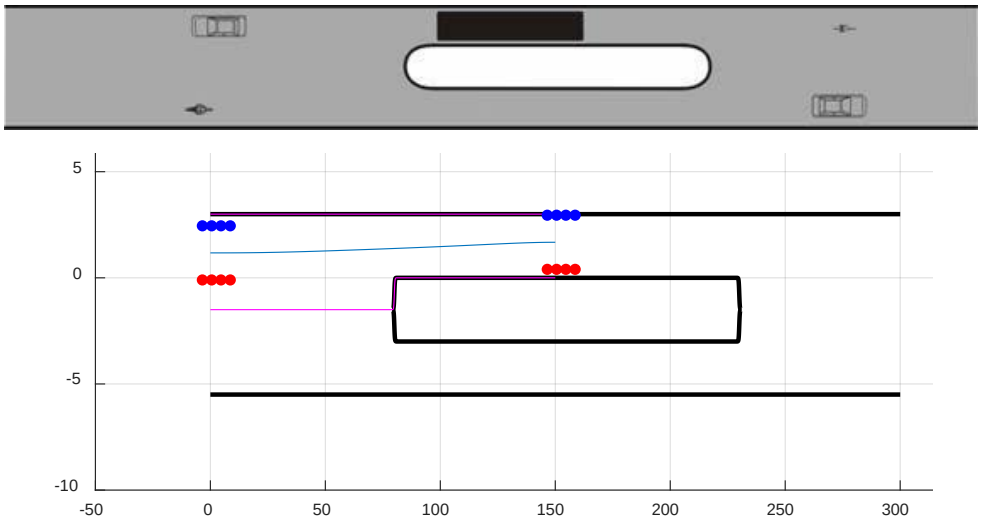


Figure 3.5: Simple one lane stop in MATLAB. Starting and ending points are shown. Left bus side showed in blue and right side in red. Boundaries in magenta [17].

- Bus stop that is separated from a highway. (Here both up and low geometry constraints are based on the equation (3.3) with different values to define a path that goes outside the highway).

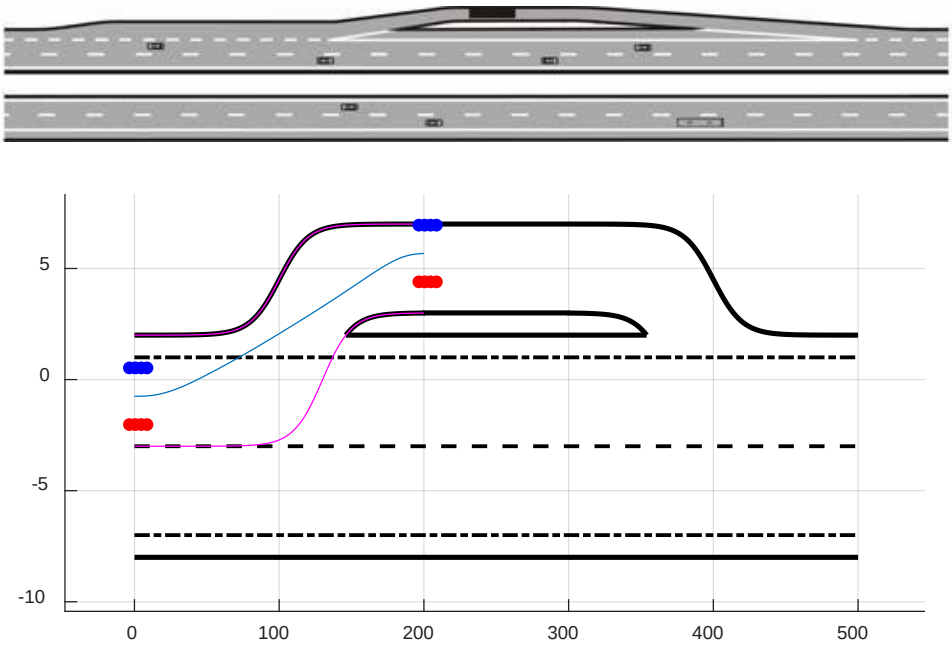


Figure 3.6: Bus stop next to a highway. Boundaries in magenta. Left side of the bus in blue and right side in red [17].

This method works for scenarios where the sidewall profile can be approximated by a unique continuous function that defines the bus stop geometry. To introduce the problem of implementing a real bus stop in the algorithm, given a drawing with the measures that define its geometry, the following real bus stop that will be used in the simulator and later on in a real test is considered:

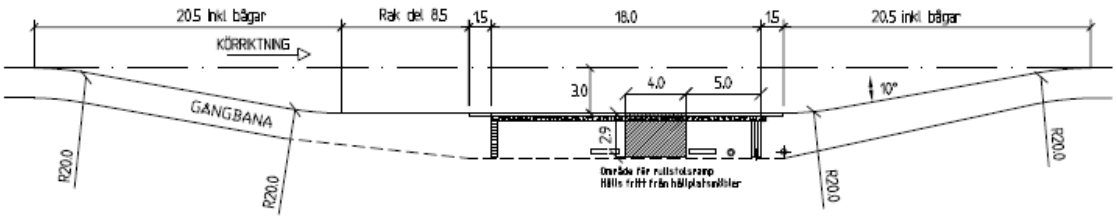


Figure 3.7: Real design of a bus stop used at the real test.

The objective is to set the low geometry boundary $e_{y(\min)}$ at instant k for the N_l points of the bus as shown in Figure 3.8 (the upper boundary $e_{y(\max)}$ is just a constant numerical value that will denote a straight line). Note that the only data available from this drawing are the coordinates of points that define the geometry of the sidewalk and there are no known functions that define this geometry design and let remember that the positions (x_p, y_p) of the points defining the bus side evaluated at each instant k are symbolic before the solution of the OCP is computed (eq. 3.5-3.7).

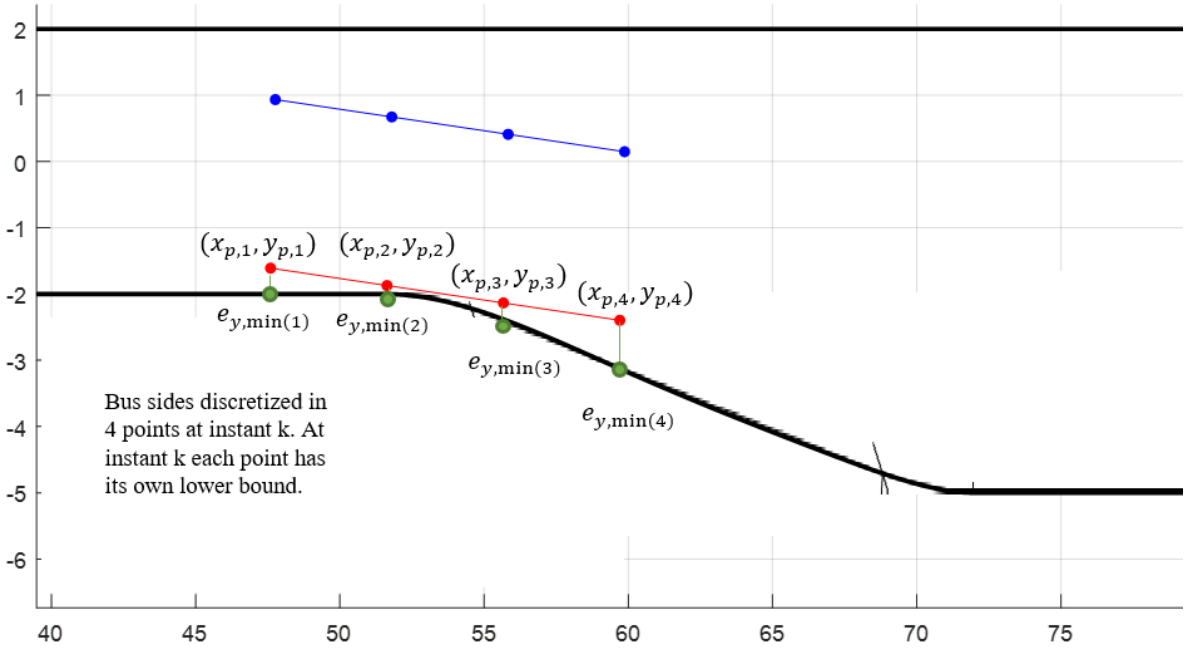


Figure 3.8: Representation of the problem of finding the boundary constraints e_y for each one of the Nl points that define the bus side at instant k.

Different approaches were used to solve this problem. The approach of having an array of points e_y that define the bus stop is not valid since at instant k, $e_{y,k}$ the same constraint would be defined for all the points that define the bus side (which is not true

as shown in Figure 3.8, the value of $e_{y,k}$ is different for each point at instant k). Other approach is to use the use of interpolators which are able to work with symbolic variables and CasADi has in-built functions to this kind of interpolators like *interpolant*. Then, by introducing the x and y coordinates of all the points of the bus stop, CasADi can interpolate the x_p variables to obtain the corresponding e_y values. However, this method introduces discontinuities resulting in a non-differentiable constraint that makes the NLP solver fail when computing the solution. Another approach, trying to define the bus stop by a continuous and differentiable function, was to find the values for the expression (3.3) that make this function to fit best to the points of the bus stop. A nonlinear curve-fitting using a nonlinear least-square solver *lsqcurvefit* [18] in MATLAB was used, returning the values of $a, x0S$ that best fit the data, giving the approximation of Figure 3.9, which is not very accurate (the bus can hit the sidewalk and go over it):

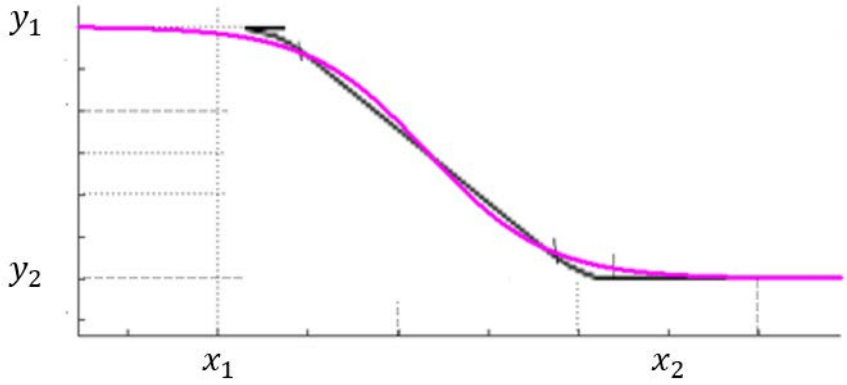


Figure 3.9: Bus stop approximation using a least square curve fitting for the function (3.3) to data defining the geometry of the bus stop.

Other approaches like using polynomial fitting were discarded due to the need of using high degree polynomials which suffered from overfitting problems which make CasADi fail in solving the OCP. The final solution adopted was using an in-built CasADi function *if_else* that allows the use of conditionals using symbolic variables. This allows the use of a piecewise function to describe the bus stop, thus, depending on the symbolic $x = (k-1)ss + x_p$ the adequate function for the corresponding interval will be evaluated. The piecewise polynomial is thus defined by:

$$e_{y,(min)}(x) = \begin{cases} y_1 & x \in [0, x_1) \\ p(x) & x \in [x_1, x_2] \\ y_2 & x \in (x_2, Sf] \end{cases} \tag{3.11}$$

where $p(x)$ is a polynomial that fits the data points in the interval $[x_1, x_2]$. To do the polynomial fitting, instead of using existing solutions like the MATLAB function *polyfit*, a least-squares fitting was solved in CasADi which allows the implementation of constraints at the extremes to ensure continuity (and for the derivative at the extremes too) and a constraint for not allowing the polynomial to go beyond the actual limits of the bus stop:

$$p(x_1) = y_1; \quad p'(x_1) = 0; \quad p(x_2) = y_2; \quad p'(x_2) = 0 \tag{3.12}$$

The result, compared to the polynomial generated by the function *polyfit* is shown in the following figure:

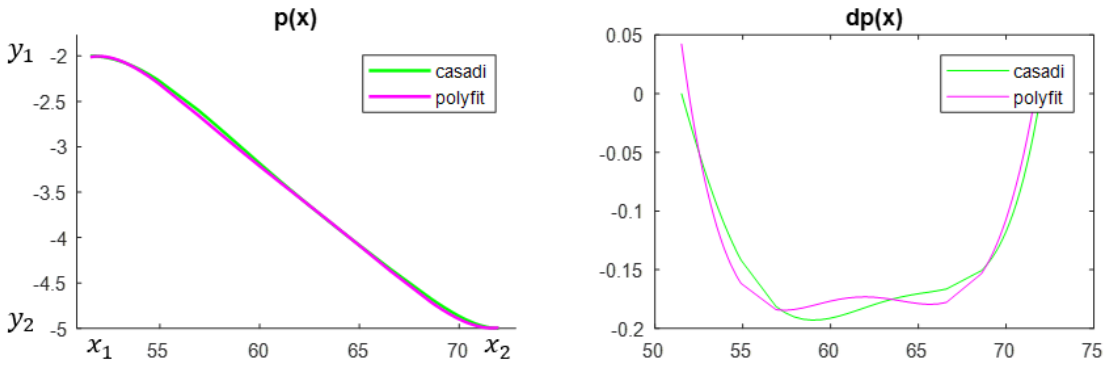


Figure 3.10: Polynomial fitting comparison between *polyfit* and CasADi with constraints. Both polynomials evaluated in a defined interval (left) and their derivatives (right).

They look similar, but the at the extremes, the *polyfit* polynomial does not start/finish at the desired points and the derivative at the extremes is not equal to the derivative of the other two functions that define the bus stop. The complete bus stop geometry constraint is then completely defined and an exact representation of every kind of bus stop can thus be defined. Note that in order to perform a right bus stop, the final constraint that defines the final e_y position has been changed.

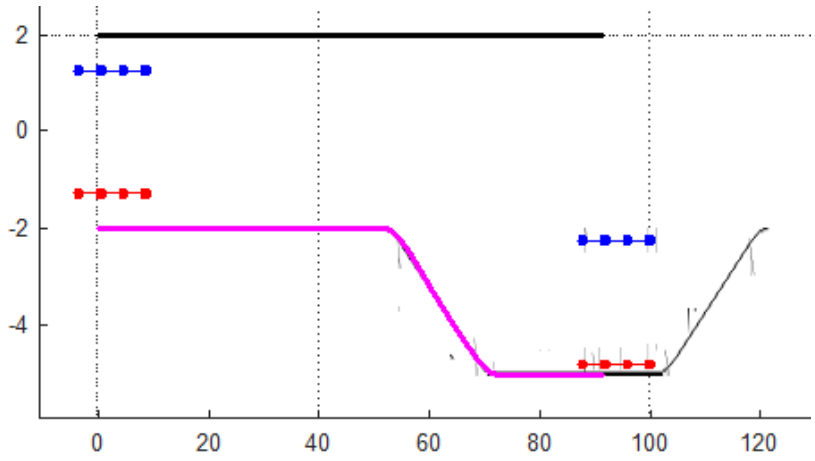


Figure 3.11: Right side bus stop geometry (magenta). Initial and final positions of the bus are shown. Left bus side is shown in blue. Right bus side is shown in red.

3.3.2 Take-off problem

Before it was shown how the geometry boundaries and final position of the bus constraints could be modified to have different bus stop scenarios. It is easy to see how changing other constraints, gives the possibility of having more scenarios like a bus take-off or driving along a path.

To allow the bus to take-off, firstly the initial conditions for the states should be changed for the final values obtained in the solution of the bus stop. Hence, the initial speed of the bus is $v(0) = 0\ km/h$ and the previous final speed constraint $v(N+1) = 0$ can be removed. If a specific pose of the bus at the end of the horizon N is not required, the constraints for the lateral displacement, steering and orientation can be removed too. A new constraint for the boundaries has to be defined representing the continuation of the road following the same method as in previous section. The constraints for maximum and minimum speed, lateral acceleration, jerk are kept. Figure 3.12 shows the initial and final points of the bus for the problem of a bus stop docking with a take-off.

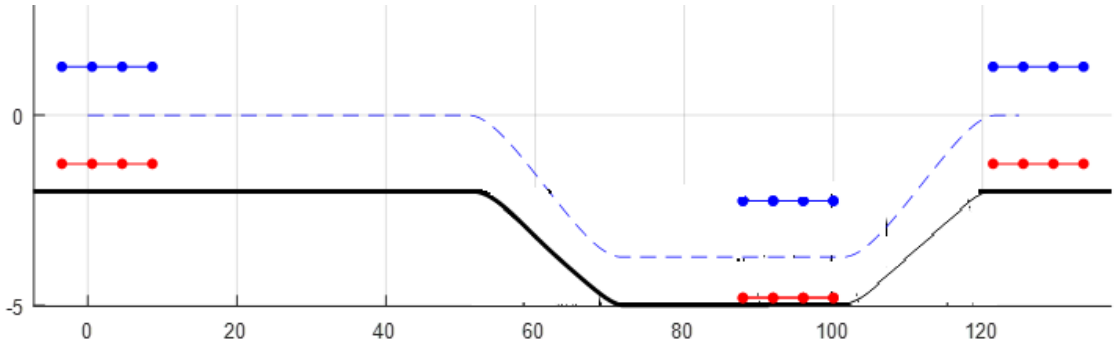


Figure 3.12: Bus top followed by a take-off- Dashed line is a reference path.

3.3.3 Circular path

Until now, only scenarios where the local curvature κ_σ of the centreline of the path σ is equal to 0 (i.e. straight paths) have been considered, according to the spatial transformation explained in 2.1, where the velocity of the vehicle along the path was defined by $\dot{s} = \frac{1}{1-\kappa_\sigma \cdot e_y} \cdot v\ cos(e_\psi)$. It is of interest to add the possibility of performing curvilinear trajectories as it is a requirement in almost all the bus networks of cities. This can be easily added to the OCP problem by passing the values of κ_σ for each interval of the N+1 grid points which discretize the path. The value of κ_σ, known the radius of the curve of the centreline of the path ρ_σ is calculated as $\kappa_\sigma = 1/\rho_\sigma$. Then, a small modification in the CasADi implementation is done by setting the known values of κ_σ as a parameter value (in the same way as for example the trajectory reference is sent). Thus, an array of N elements containing the known values of κ_σ will be a new input for CasADi which will use them when solving the OCP as shown below.

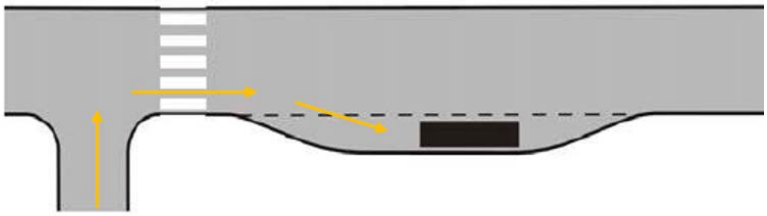


Figure 3.13: Example of a bus stop after turning to the right [17].

```
nlp = struct( 'x', V, 'p', [Vr;KAPPA], 'f', cost, 'g', g);
%x is optimization variables, p known parameter vector, g constraints;
KAPPA is vector of local curvatures for each grid point.
solver = nlpsol( 'solver', 'ipopt', nlp, ); %Build nlp solver
```

In the following, an example has been made of a scenario where an autonomous bus, starting at an initial point, has to make a bus stop docking and then take-off and turn around to come back to the starting point. The vector of κ_σ will be filled by zeros in the straights and with the value of the non-zero local curvature of each point in the four curves. The centreline of the whole path is shown in the following figure:

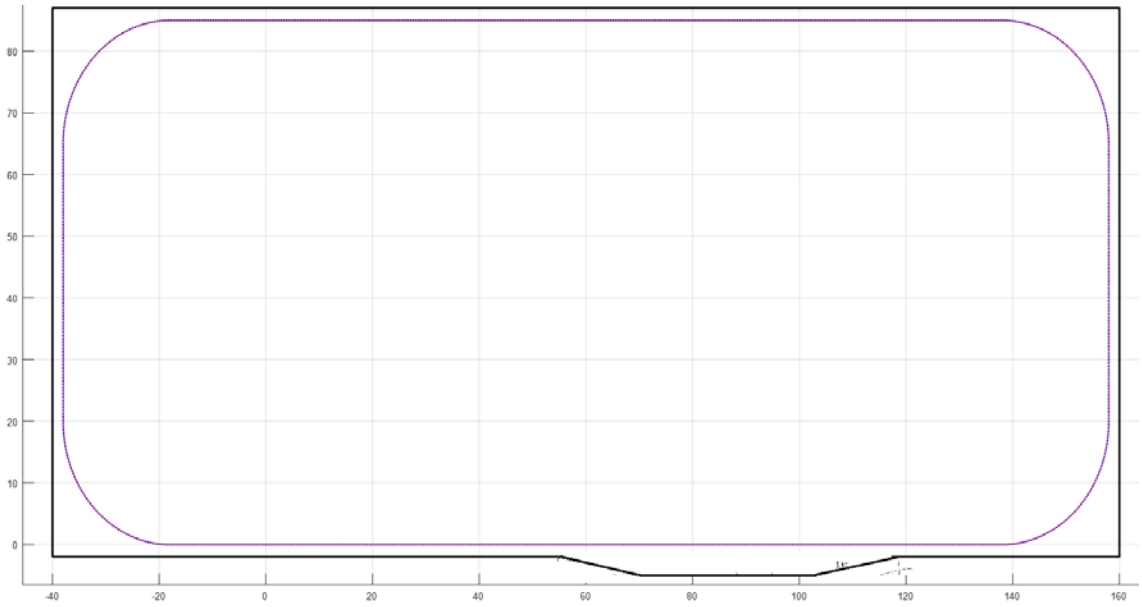


Figure 3.14: Circular path with a bus stop and four turns.

As always, the path is discretized in N points. At the desired point k, the necessary constraints to perform a bus stop should be set (i.e. lateral displacement e_y, velocity $v = 0$, orientation and steering constraints to have the bus aligned with the bus stop, etc.). The same constraints are established for the final point (same as start). For the rest of the trajectory, the usual constraints are taken into consideration.

Two solutions could be adopted when solving this scenario. One consists in computing the whole trajectory as a unique OCP. Then a continuous trajectory will be retrieved from CasADi but its computation time will be quite large because as the path is longer now than for the case of a simple bus stop, the number of grid points N should be increased to maintain an acceptable resolution. To avoid the high computation times of the whole trajectory, the second approach that consists on splitting the trajectory in smaller sections (i.e. the bus stop, the take off and the rest) can be used. The user has to be careful then to define the problem correctly as the initial conditions of one sector will be the values of the solution at the end of the previous section. Apart from the initial conditions, new constraints should be written in the program for the controls, as their initial values should be the same as the final values they had in the previous sector. This will guarantee that the trajectory obtained is continuous.

3.3.4 Time optimal trajectory formulation

In chapter 3, looking at the trajectories generated from the solution of the OCP for some situations like for example a take-off where the initial velocity is zero, the solution showed a bus that took a considerable time in reaching an acceptable speed for driving in an urban area (around 30-50 km/h) as there are not constraints for time. For the case of a bus that approaches a bus stop, but with a low initial speed, the bus made a very small increase of speed before braking for stopping, doing the bus stop slowly. It might be of interest to see some ways of solving the problem of slow trajectories that could be done faster without impairing the comfort of the passengers.

One way of putting more emphasis in speed when solving the OCP, obtaining a trajectory that will show a higher average speed, is done by modifying the values of the weights of the matrices Q and R from the cost function defined in eq. (2.21). Concretely, by setting a higher value (penalizing more) of the weight related to velocity. However, in case the weights of the matrices Q and R should not be modified, another approach modifying the cost function can be implemented.

Taking the cost function defined in equation (2.20), supposing that the minimum time t^* is known beforehand, then a constraint $t(N+1) = t^*$ can be imposed to set the final time to be the optimal one. This is equivalent to move this constraint to the cost function with a variable $\rho \cdot (t(N+1) - t^*)$. As $\rho \cdot t^*$ is a constant, then the cost function can be:

$$\min_{x,u} \sum_{k=0}^{N} \ell(\xi, u) + M\big(\xi(N+1)\big) + \rho \cdot t(N+1) \tag{3.13}$$

where $t(N+1)$ is the time at the end of the horizon and ρ is a weighting value. By adding this term to the cost function, the time that it takes to arrive to the final point is minimized and if the value of ρ is large enough, the trajectory will be equivalent to time optimality. Apart from the benefits of avoiding the bus going slow unnecessarily, the use of this approach seems to help the NLP solver to converge faster in complex scenarios. For instance, for the scenario of the circular path trajectory of the previous section where there are different constraints in the middle of the path and the value of the grid points is large, by implementing this approach the solver can find a solution but using the first approach or not using any of these two approaches the solver cannot converge into a solution.

The way of implementing it in CasADi is similar to the case of adding the vector of local curvatures shown before. In this case, a new known parameter called *rho* is introduced along with the reference and the κ_σ vector. The final term of the cost function has to be modified as shown below:

```
cost = cost+ell(V(iV('x',k)),V(iV('u',k)),Vr(iV('x',k)),Vr(iV('u',k))); % L
cost = cost + ellF( V(iV('x',N+1)), Vr(iV('x',N+1)) ); % Final cost M
cost = cost + rho*V(iV('x',N+1,'t')); % Time cost
- - -
nlp = struct( 'x', V, 'p', [Vr;rho;KAPPA], 'f', cost, 'g', g);
solver = nlpsol( 'solver', 'ipopt', nlp, opts ); % CASADI %
```

3.4 Functional Development in Robot Operating System (ROS) and Gazebo.

The high investments in developing the technologies used in autonomous vehicles make that new ways of testing the algorithms, sensors and actuators which are used in these vehicles arise to make the process of testing cheaper and faster. The use of simulators which use accurate dynamic models of the vehicles and the sensors that are installed in autonomous vehicles, are a good choice and their use is increasing continuously. This is the case for the autonomous bus used in the project, where a simulator based on Robot Operating System (ROS) and Gazebo is being developed. The understanding of the architecture which forms the simulator is of major interest for this project, as the simulator allows the testing of the trajectories generated in this thesis without the need of using the real bus. Furthermore, for the continuation of the work done in this thesis, the simulator will make testing of new algorithms easier and faster than testing in a real autonomous bus as done now.

3.4.1 ROS

ROS is an open-source, meta-operating system for robots that provides tools and libraries for obtaining, building, writing and running code across multiple computers. It can also be described as a middleware that provides services to software applications beyond those available from the operating system, making it easier for the software engineers to focus on the purpose of the application, as they have more facilities to perform the communication input/output. ROS is not an actual operating system nor a programming language or programming environment [19]. ROS based running processes can be represented in a graph as in Figure 3.15 [19]. where the processing is performed in *nodes* that can send and receive *messages* which are sent through a *topic*.

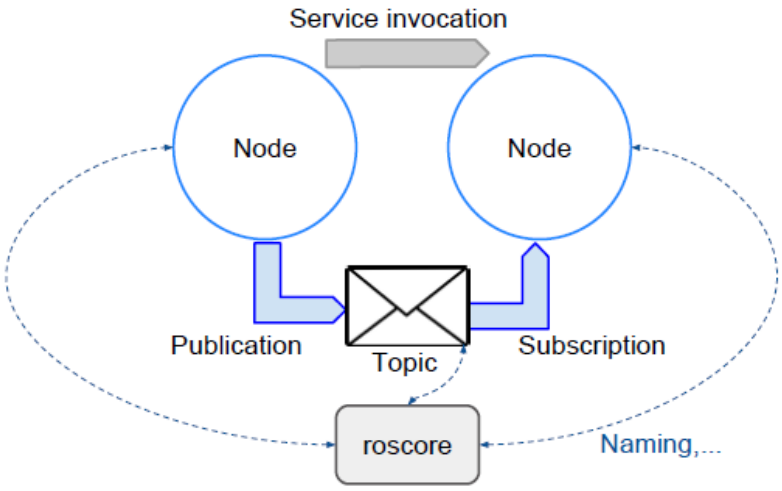


Figure 3.15: ROS graph architecture [19].

The main elements that can be found in ROS architecture are:

- Nodes: are processes that performs computations. If a failure takes place, it is isolated to the individual node that fails. They have a name that allows its identification by the other nodes.
- Message: is a simple data structure that allows the communication between nodes. There are different types of messages that go from basics like floats, integers or Booleans to more complex structures that can contain arrays, strings. geometry poses…
- Topics: can be defined as buses which are used by the nodes to exchange messages Nodes can subscribe or publish to a topic depending on if they want to send or receive messages from that topic.
- Service: a node gives a service that can be requested at any time. It blocks the client node until the reply is delivered.
- Action: a node requests a goal and this gives feedback or the current status until the result is delivered. It does not block the client node during the action.
- Master: enables different nodes to locate each other by providing a naming and registration service

ROS provides also other useful tools like *rviz* which a 3D visualization tool for visualizing topics. This means that the user can see a representation of the information that go through the topics like data from sensors:

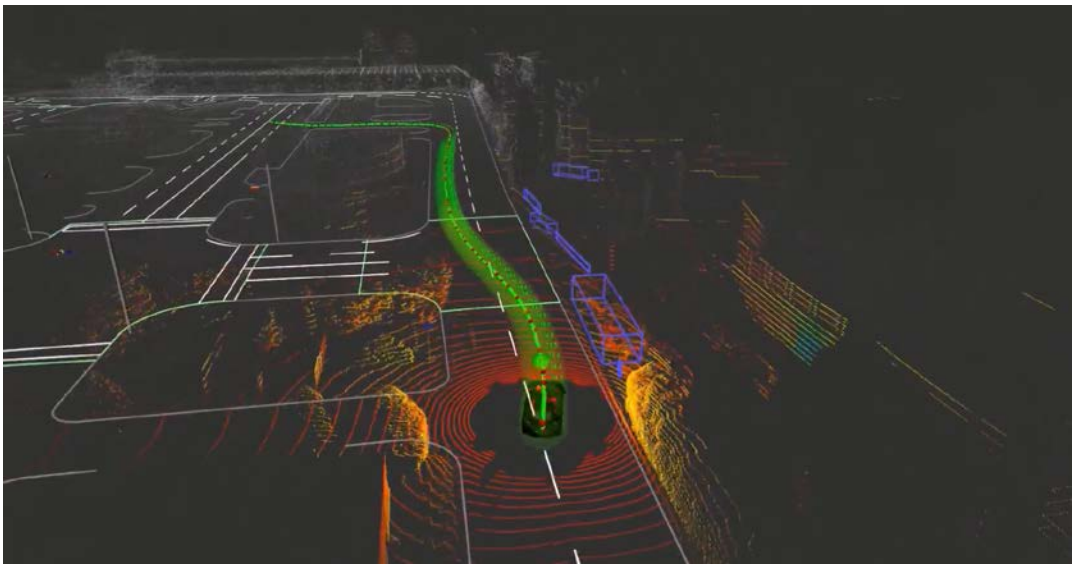


Figure 3.16: 3D visualization of the sensor readings of an autonomous vehicle and the followed path in rviz [19].

3.4.2 Gazebo

Gazebo is a 3D dynamic simulator that allows the testing of robotics algorithms and design of robots in different environments, offering multiple physics engines with high degree of fidelity, possibility of designing different sensors, and provides libraries and

programming and graphical interfaces. It has a great integration with ROS which serves as the interface for the robot [20]. This means that the user can use the same code in Gazebo than in the real physical robot making it faster to test a robot in a simulated environment than setting a real scenario for a real robot with the associated time and costs.

Worlds (environments) can be created using the graphical interface that Gazebo has. There exist lots of pre-built models of different objects, buildings to create them but there are also pre-built robots which incorporate all the sensors and actuators needed for testing. A 3D representation of the right bus stop shown in section 3.3.1 which will be used in the real test has been built using some walls, but more complex scenarios can be created using tools such as Open Street Map along with Blender, as can be seen in the representation of Lindholmen (Gothenburg) shown in Figure 3.17 [5].

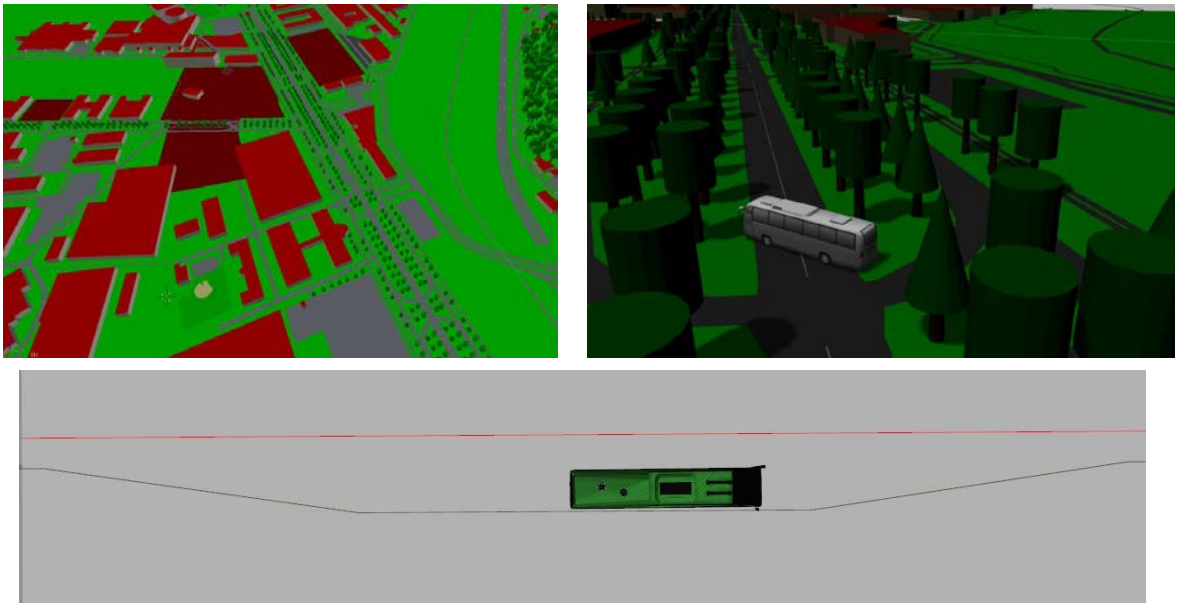


Figure 3.17: 3D representation of Lindholmen (Gothenburg) in Gazebo using Open Street Map (top left) converted to a Gazebo world file with a bus model (top right) and a simple design of the right bus stop in Gazebo (bottom) [5].

Furthermore, a model of the autonomous bus is available. It has the actual measures of the bus, but also incorporates physical properties such as the mass of the bus, wheels, inertia moments etc.

Figure 3.18: Gazebo model of the autonomous bus.

3.4.3 Simulation architecture [5].

ROS and Gazebo are software which are used inside a bigger architecture that forms the simulator that is being developed at Volvo AB including an environment and a complete vehicle model. Some time was spent in understanding how this architecture works and small collaboration was done in its development (providing help in some moments or creating a script to export to the simulator the trajectory obtained from the solution of the OCP). This has been helpful to understand how the process of developing a simulator tool for autonomous platforms is.

The architecture of the simulator can be divided in several subsystems which are:

- Vehicle Environment Management

This subsystem includes sensing, mapping and localization and it can sense the surroundings of the vehicle thanks to its sensors and by means of sensor fusion algorithms (using data from, for example, Inertial Measurement Units -IMUs- or Global Positioning Systems -GPS-), simultaneous localization and mapping can be done to allow the localization and position of the vehicle in the map.

- Virtual Transport Model (VTM)

This subsystem includes all the elements needed to govern the vehicle motion. These elements are other subsystems that perform different functionalities and are described as follows:

  - Route Segment Management and Route Situation Management: it manages the route that has to be followed by the vehicle and gets the information of a file that contains the path information at different time steps (spatial and orientation coordinates along with velocities and time information). This information is sent to the Traffic Situation Management subsystem.

  - Traffic Situation Management and Vehicle Motion Management: it includes traffic situation manoeuvres and prediction based on the information received from the route. It sends a steering angle δ and speed v_x request to the Vehicle Motion Management subsystem which communicates with the vehicle plant that has a detailed dynamic model (steering, suspension, axles, wheels, etc.) of the bus. This model takes the longitudinal and lateral dynamics of the bus given steering angle and velocity request from the path planner. It also implements a linear tire model with combined slip among others. Depending on the request and the states, this subsystem sends the necessary control outputs to the Motion Support Device Management subsystem through Controller Area Network (CAN) bus.

  - Motion Support Device Management: is the interface between the controllers and the mechanical vehicle plant, including all the actuators of the bus like propulsion, transmission or brake systems.

The Vehicle Environment Management and Route Segment and Situation Management are implemented with ROS and it is able to communicate with the VTM architecture, and receives the position, orientation and velocity information of the trajectory that is wanted to be followed. With Gazebo, and thanks to its physics engines and the model of the bus, a visualization of the simulation is obtained while at the same time it sends feedback of the bus pose to the ROS architecture, closing the loop of the whole architecture. Figure 3.19 shows a flow chart showing how the subsystems mentioned above are connected:

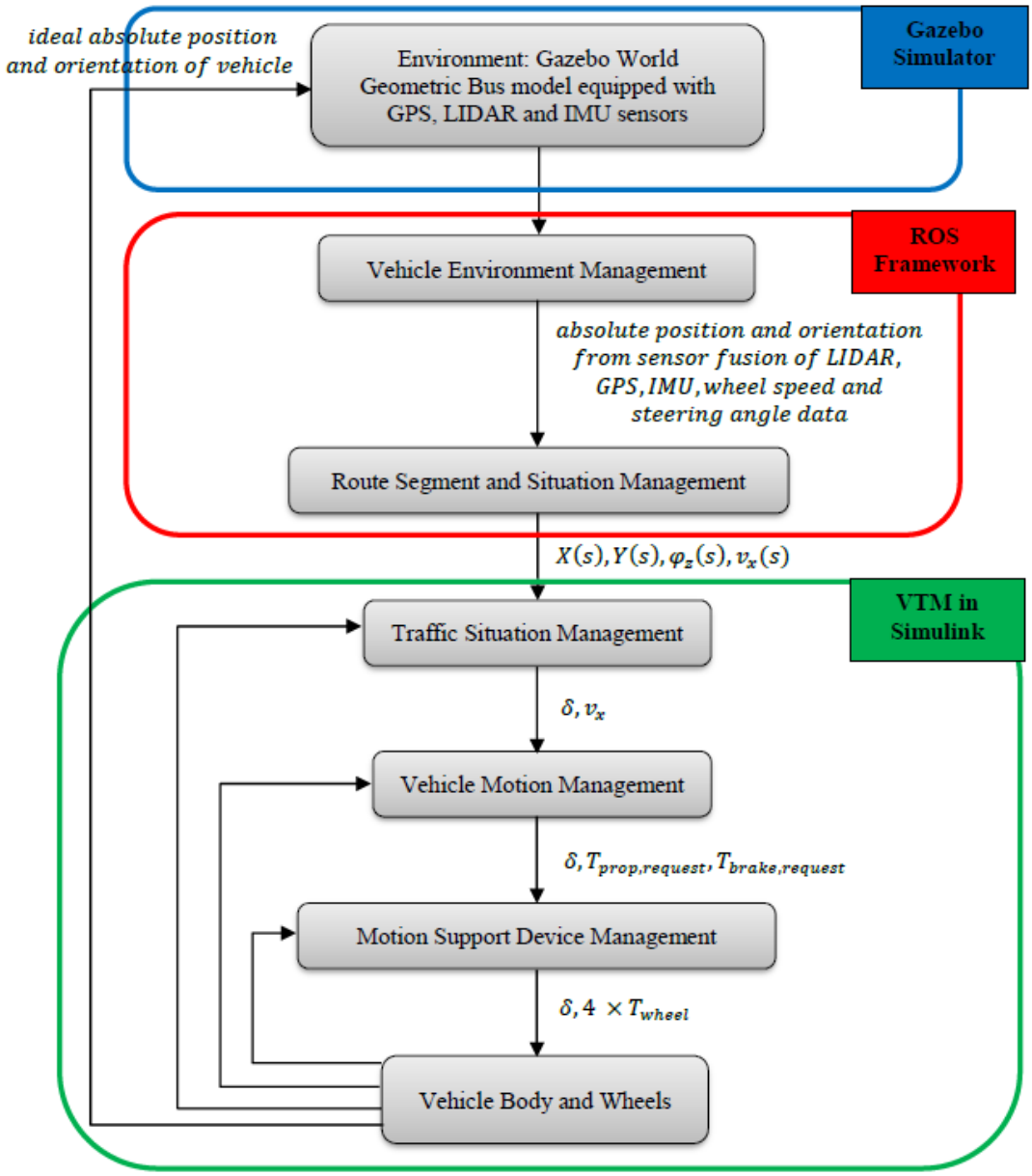


Figure 3.19: Simulation architecture developed and implemented at Volvo AB for the autonomous bus project.

The VTM which is implemented in Simulink was successfully exported as a standalone C++ ROS node, being able now to have it working without using MATLAB, increasing the overall performance. The path follower, which is included in the Traffic Situation Management, has been tested successfully by Volvo with a real autonomous bus in a bus stop scenario and it is being adapted and connected to Gazebo simulator to allow the testing of new trajectories in it without having to use the real bus each time. Gazebo will take the steering angle and velocity requests (as the real bus does) in order to move the bus model. Gazebo is connected to the ROS framework, closing the loop, in such a way that with the feedback given by Gazebo, the path follower can find the bus position along the trajectory and perform the calculations needed to continue following the path. However, this stage is still being developed and it does not work correctly yet, resulting in the bus being able to follow the trajectory at the beginning but it loses it at some points, being unable to reach the goal point, but as it was mentioned at the beginning of this section, having the simulator working correctly will be an essential tool for the continuation of the work done in this thesis.

3.4.4 Exporting the OCP solution trajectory to ROS.

The solution of the OCP is composed by an array that contains the information of the values of the states and controls along the path that makes the bus able to complete the trajectory which meets the required constraints. The Route Segment Management and Route Situation Management subsystem needs this array in a special format in a .xml file with the information of the spatial and orientation coordinates of the bus in the global frame, speed and time. Other fields are needed like the identifier of the Gazebo map or the number of points in which the path has been divided among others. A MATLAB script called *writeXMLfile* was created to export the OCP solution to a .xml file that the simulator architecture can understand like the one shown below:

```
<path>
   <info date="06/04-2018" entries="1001" map_uuid="6f8ad05ee41c" recorded_in="simulator" />
   <entry id="1" pitch="0.000000" roll="0.000000" speed="0.277778" time="0.000000" x="0.000000"
y="0.000000" yaw="0.000000" z="0.000000"/>
   <entry id="2" pitch="0.000000" roll="0.000000" speed="0.811538" time="1.157821" x="0.527695"
y="0.000001" yaw="0.000005" z="0.000000"/>
   <entry id="3" pitch="0.000000" roll="0.000000" speed="1.282949" time="1.662195" x="1.055390"
y="0.000005" yaw="0.000011" z="0.000000"/>
   <entry id="4" pitch="0.000000" roll="0.000000" speed="1.626062" time="2.024946" x="1.583085"
y="0.000013" yaw="0.000017" z="0.000000"/>
   <entry id="5" pitch="0.000000" roll="0.000000" speed="1.907222" time="2.323602" x="2.110779"
y="0.000022" yaw="0.000020" z="0.000000"/>
           .                                       .                                       .
           .                                       .                                       .
           .                                       .                                       .
   <entry  id="998"  pitch="0.000000"  roll="0.000000"  speed="1.739618"  time="102.384687"  x="-
1.583085" y="0.000000" yaw="-0.000000" z="0.000000"/>
   <entry  id="999"  pitch="0.000000"  roll="0.000000"  speed="1.423891"  time="102.718252"  x="-
1.055390" y="0.000000" yaw="-0.000000" z="0.000000"/>
   <entry  id="1000"  pitch="0.000000"  roll="0.000000"  speed="1.012207"  time="103.151420"  x="-
0.527695" y="0.000000" yaw="-0.000000" z="0.000000"/>
   <entry   id="1001"   pitch="0.000000"   roll="0.000000"   speed="0.277778"   time="103.981621"
x="0.000000" y="0.000000" yaw="0.000000" z="0.000000"/>
</path>
```

4

Results

4.1 Solution of the Optimal Control Problem.

Consider the initial problem from section 3.2 of an autonomous bus that approaches to a bus stop located at the left side of a road. The distance from the initial point to the final point is $S_f = 100m$. The initial conditions for the states $\xi_t = (e_y, v, a, \psi, \delta)$ are set to 0 except for the initial speed which is set to $v_0 = 45\ km/h$. The values of the matrices Q, R and P for the cost function in equation (2.20) are set with the following weights:

$$Q = \begin{bmatrix} 10^{-2} & 0 & 0 & 0 & 0 \\ 0 & 10^{-2} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \quad R = \begin{bmatrix} 1 & 0 \\ 0 & 10 \end{bmatrix}; \quad P = Q \tag{4.1}$$

The dimensions of the bus can be found in the datasheet of the bus used (appendix A). The number of points N_l in which the side bus is discretized is $N_l = 4$ and the number of points in which the path is discretized has been $N = 200$ as it gives a resolution of $ss = = \frac{S_f}{N} = 0.5$. The constraints for the maximum and minimum longitudinal acceleration, jerk and lateral acceleration are set initially to $a_x \epsilon [-1,1]\frac{m}{s^2}$; $b \epsilon [-1,1]\frac{m}{s^3}$ and $a_y \epsilon [-1,1]\frac{m}{s^2}$. Final velocity has been set to be $v(S_f) = 1\ km/h$ to avoid the singularity when doing the transformation to the spatial state. The final position of the bus has to be 5 cm away from the sidewalk at distance S_f. Once that the problem constraints have been set, the algorithm is executed getting the following solution:

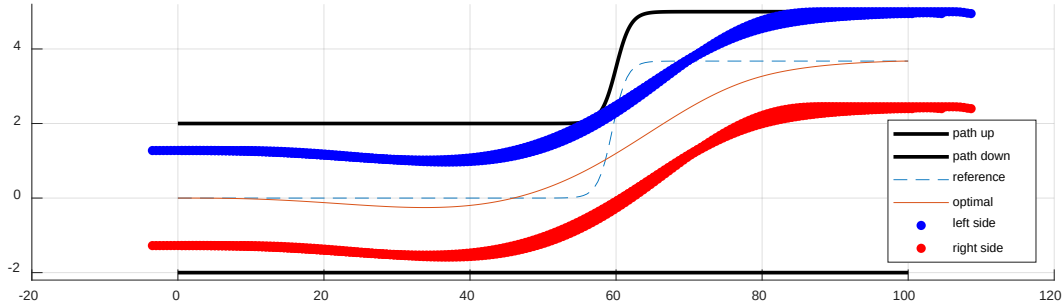


Figure 4.1: Optimal path solution for a left side bus stop.

In Figure 4.1 the optimal path defined by the optimal values for the lateral displacement e_y along the grid of N+1 points which were retrieved from the solution of the OCP is represented by the orange line. The dashed blue line is a reference trajectory and the blue and red lines represent the trajectory of the points that discretize the bus sides (left side in blue and right side in red). Note that the sides of the bus do not pass the boundaries of the lane due to the constraint imposed in eq. (2.31).

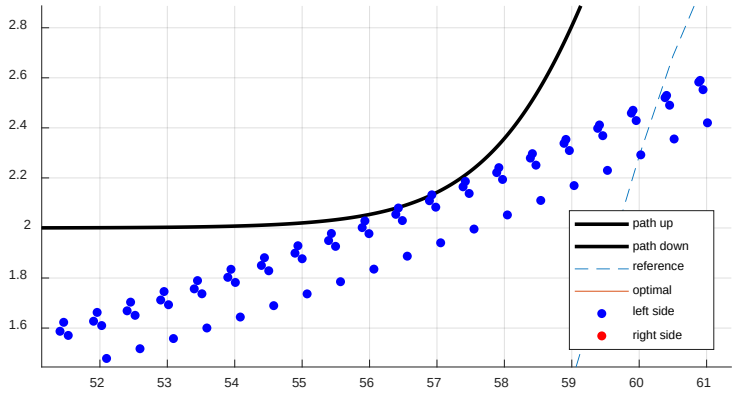


Figure 4.2: Detail of the corner. The bus does not go beyond the sidewalk.

The values for the rest of the states and control signals are shown below:

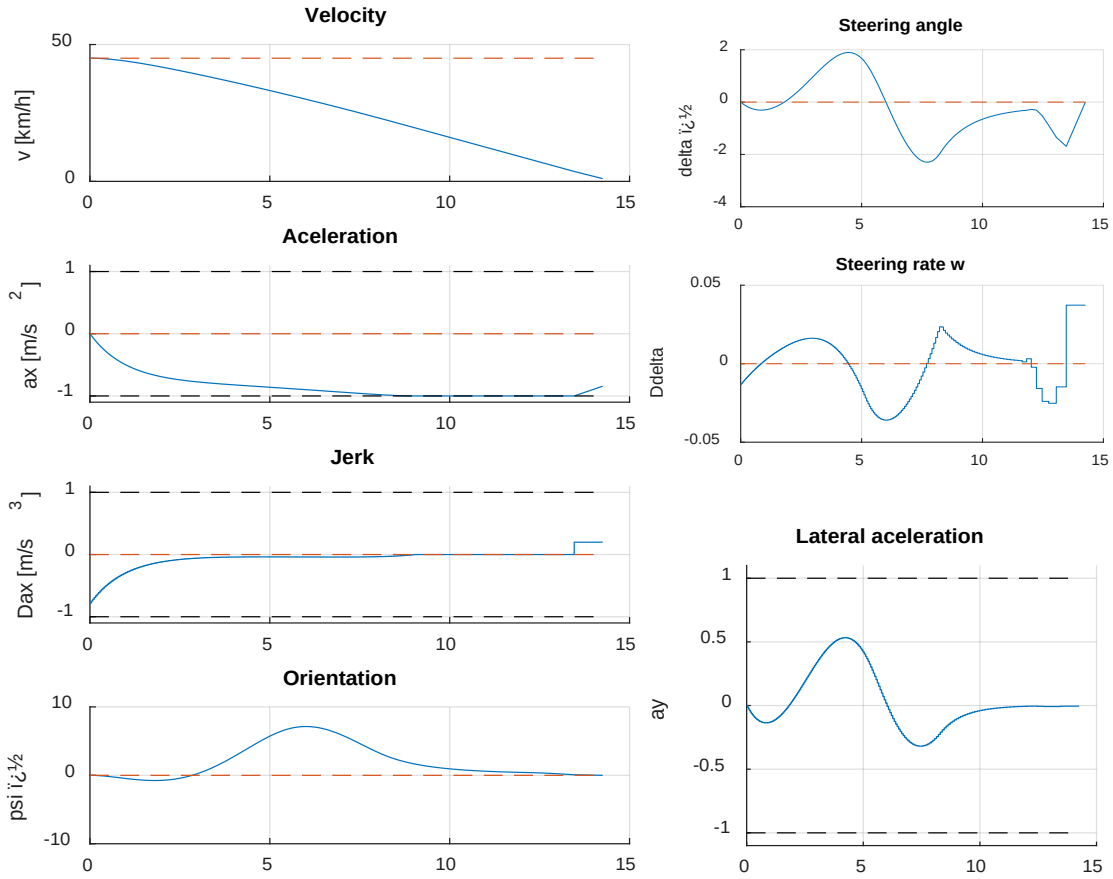


Figure 4.3: Optimal solution for the states and control signals.

As it can be seen in Figure 4.3, the constraints imposed for the longitudinal, lateral acceleration and jerk are fulfilled. Coming from a speed of 45 km/h, note how the velocity only has to be decreased to arrive to the final point and there is no need in increasing it. Additionally, the reference was set with a decreasing velocity profile. If a low initial speed would have been set instead of 45km/h and a high constant velocity reference is set, the velocity profile would have increased the speed of the bus until a certain point where it would start to decrease to perform the bus stop. This case shows how changing the reference and the weights of the matrices Q and R can lead into different solutions that could be prefered in some scenarios.

4.1.1 Other bus stop geometries solution

In this section it will be shown and briefly discussed how the optimal trajectories are found for some of the bus stop designs showed in section 3.3.1. The values of the constraints are maintained except for the geometry constraints. For example, for the bus stop which is situated next to a road the following results were obtained:

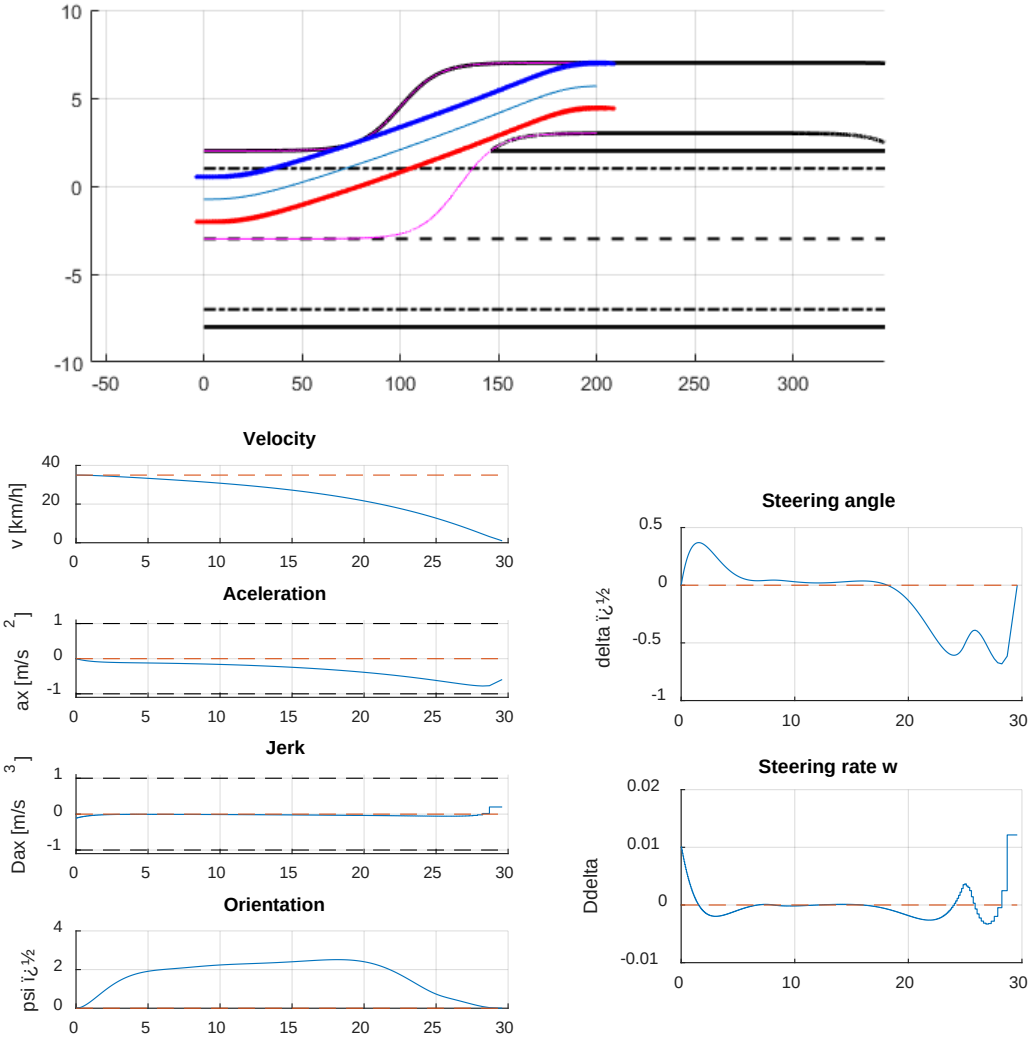


Figure 4.4: Optimal trajectory for a bus stop placed outside a road.

Note how the bus sides do not go beyond the boundaries of the path delimited in magenta. If the user prefers a solution where the bus does not go so close to the sidewalk as in the corner, the constraint in (2.31) can be modified by adding or subtracting the desired margin depending on which side of the path is considered.

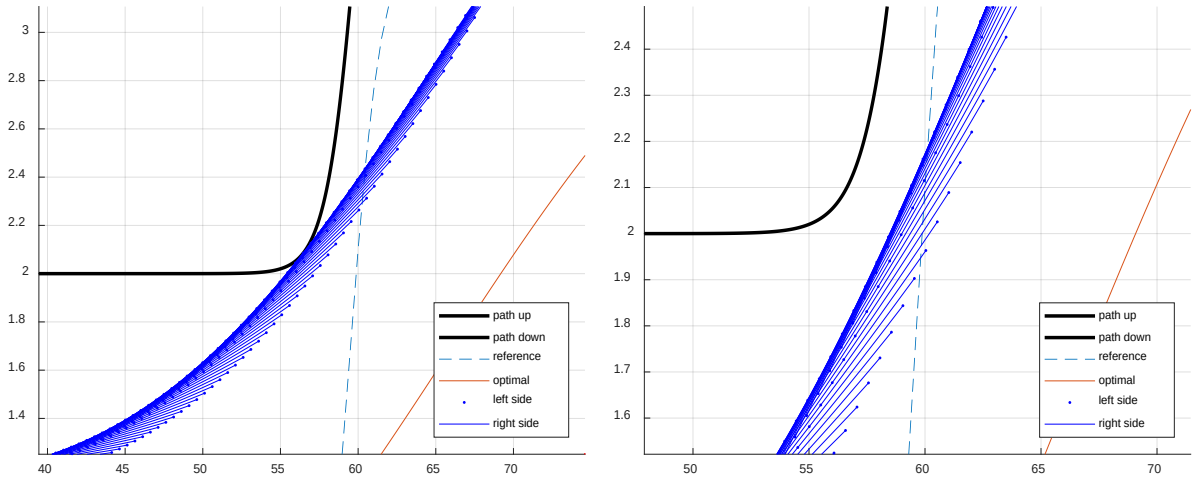
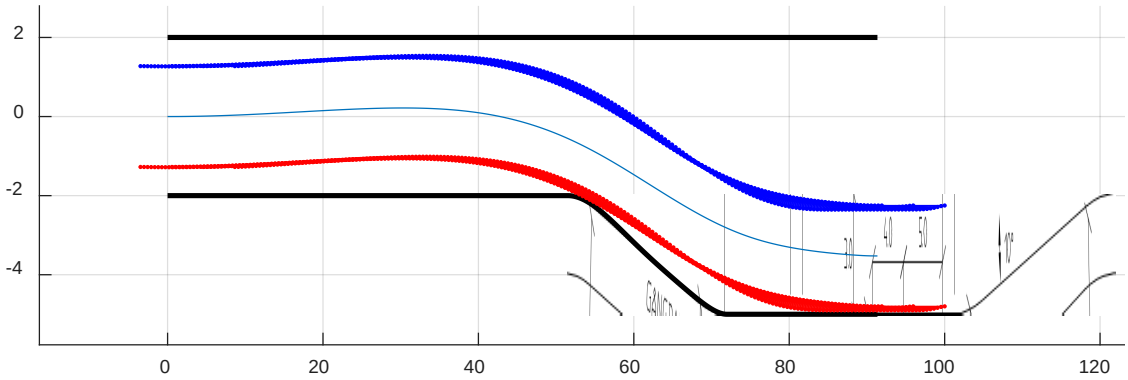


Figure 4.5: Comparison between two solutions. Constraint for not going beyond the sidewalk without margin (left) and a 0.3m margin (right).

Another example is shown using the real geometry bus stop used in the simulator and in the real test. To see some variations in the trajectory profiles, the initial velocity will be of 1km/h (to avoid the singularity). Instead of a decreasing velocity profile, for this case it has been set a constant velocity profile of 20 km/h. Maximum and minimum speed is set to be of 1km/h and 20km/h respectively along the path. The polynomial fitted to points of the bus geometry using CasADi is used to define the lower geometry constraint. Final position constrains are modified too to stop at the right side. The weights of matrices Q and R are kept.



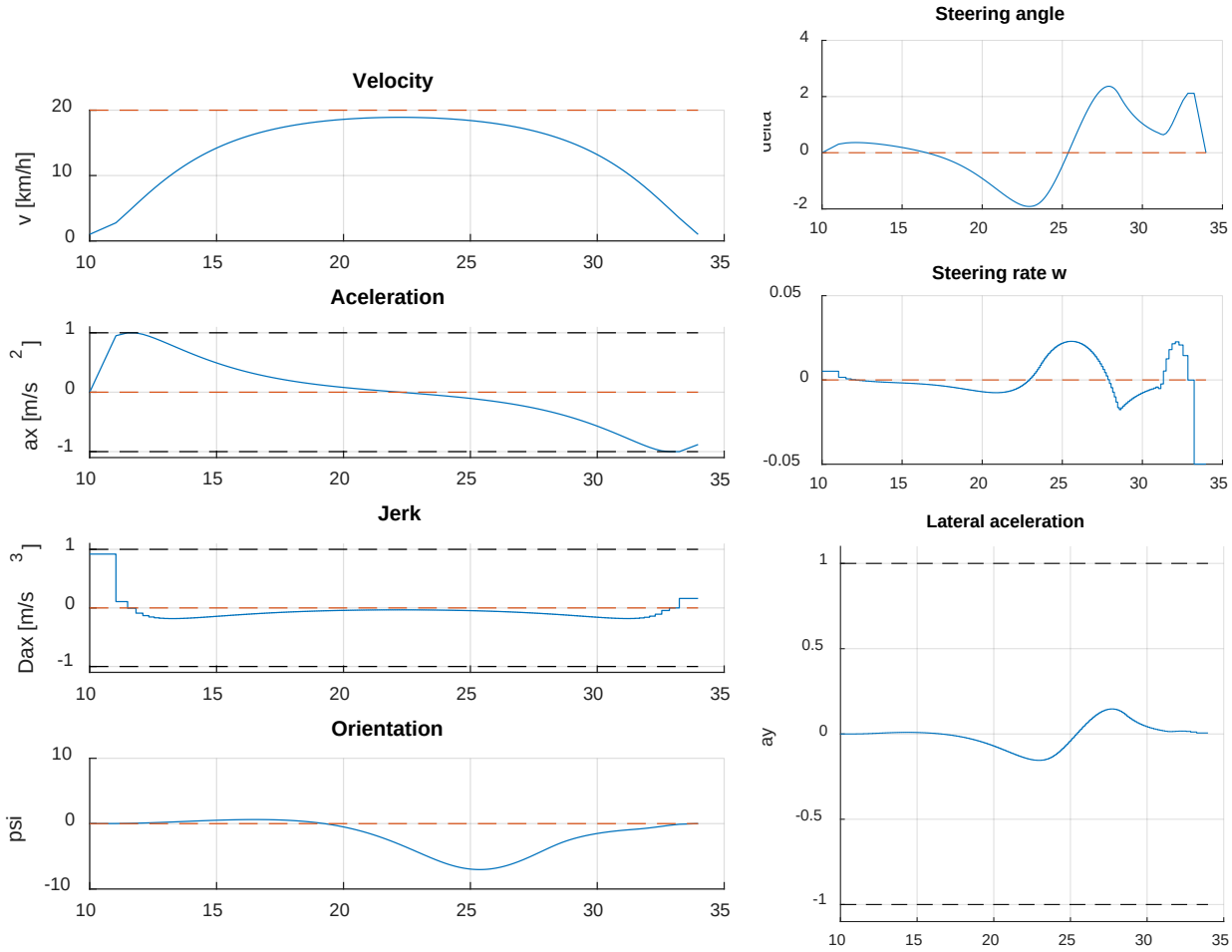


Figure 4.6: Trajectory solution for the real bus stop scenario.

As the reference for the velocity has been set to be a constant of 20 km/h, the velocity profile shows an increment of it trying to reach 20 km/h until the point where the bus has to start decreasing the speed to perform the bus stop. If the weight of matrix Q related to velocity is increased, it is expected a more aggressive acceleration that will reach the reference velocity before. Note that none of the constraints are violated along the path. Due to the low speed in this example, the lateral acceleration values are very low but longitudinal acceleration shows it maximum at the beginning when the bus takes-off and then decreases until the end.

4.1.2 Circular path solution

The problem described in section 3.3.3 is of interest because it involves the problem of a bus stop (starting from 1 km/h), then a take-off and go around to come back to the starting point. As there are 4 turns, the local curvature at these sectors should be taken into account as explained before. Different approaches can be taken into consideration for solving a problem that, due to the length of the path, if a good resolution is desired the number of

grid points N has to be high, which will result in a higher dimension problem that will be more time to solve or the solver may not converge into a solution. To avoid that, instead of solving the problem at once, the lap can be split in several sectors which can be solved faster. Three cases have been evaluated to see if there are significant differences between them:

- Solve the complete OCP at once.

The length of the centreline of the path is 527.6634m and the number of grid points has been chosen to be N=1200 to have a sampling space equal to 0.5277. Maximum and reference speeds are constant along the path and equal to 30 km/h. The front of the bus has to stop at 100m so a velocity constraint of 1km/h is set at that point along with constraints for orientation and steering angle equal to 0 (bus aligned to the bus stop). Usual values of constrains for acceleration and jerk are kept. Lane width of the path is 4m, hence geometry constrains are $e_{ymax} = 2\ m$ and $e_{ymin} = -2\ m$ except for the bus stop area where e_{ymin} is defined by the polynomials obtained before. The solution is the following one:

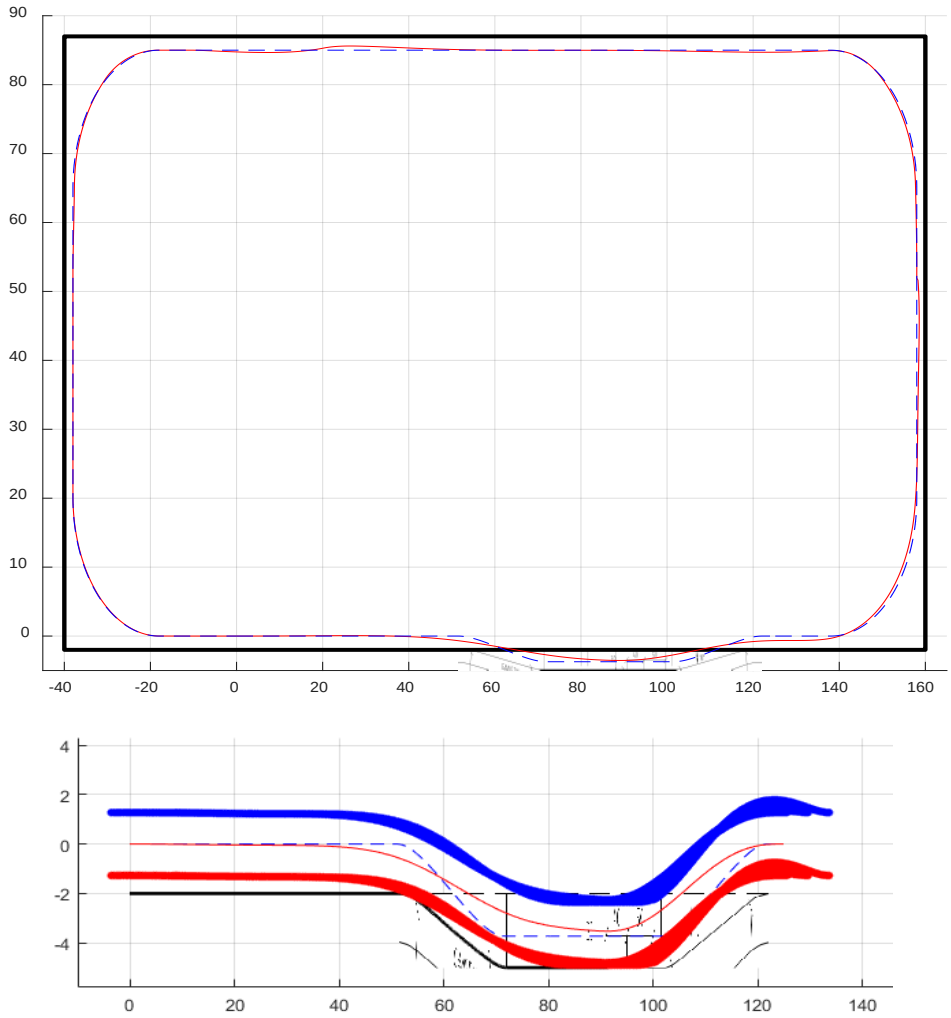


Figure 4.7: Closed loop trajectory solution (top) and detail of the bus stop section (bottom). Red line is the optimal path. Blue dashed line is the reference path. Thick blue and red lines are the left and right bus sides respectively.

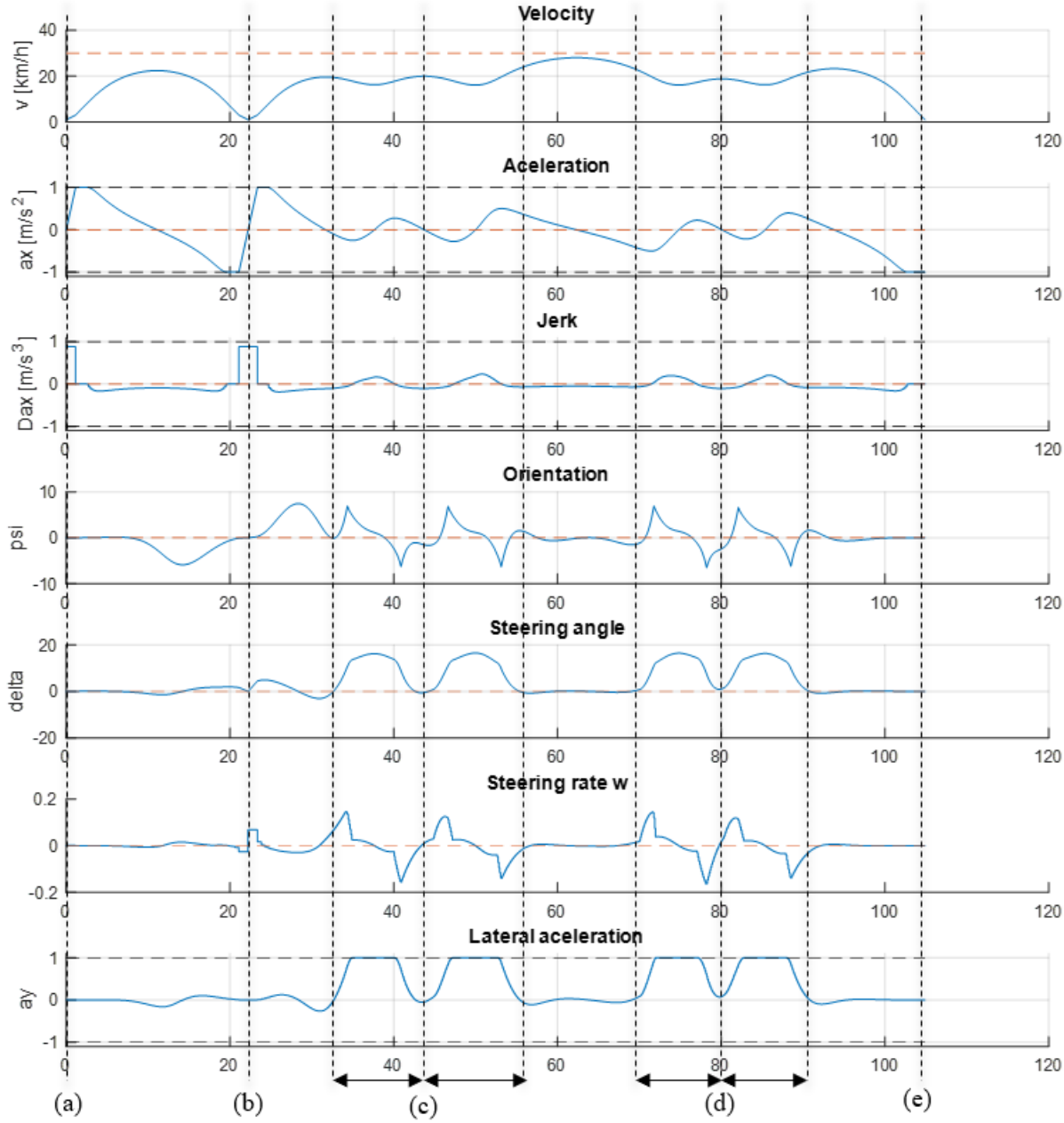


Figure 4.8: Trajectory solution of the circular path. Starting point (a), bus stop and take-off (b), first and second turns (c), third and fourth turns (d), final stop at end point (e).

Figure 4.7 and Figure 4.8 show the solutions of the trajectory generated. In previous scenarios where only straight paths where evaluated, the solution obtained in the spatial coordinates corresponded to global coordinates. Now equations (2.9) should be used to transform the spatial coordiantes to global coordinates and plot them correctly. The moments at which the bus turns to the left four times and performs the bus stop can be easily indentified. Maximum lateral acceleration takes place at left turns, maximum longitudinal acceleration in the take-off and braking situations. Elapsed time was of an average of 14.35 seconds for the NLP solver to converge into a solution.

- Dividing the path in 6 sectors.

Previous solution took a long time to be found. Now the trajectory will be sliced in 6 sectors (bus stop, take-off, and four turns) to see if by having 6 simpler problems the computation time is improved. Some aspects have to be taken into consideration: the final values of the states and control signals have to be the initial conditions for the following sector. The constraints of final speed to be 1km/h at the end of each sector can be ignored (except at the first and last sectors). With exception of the first and sixth sectors where the bus has to stop, the final position, steering and heading constraints are free. The number of samples for each region was N=200.

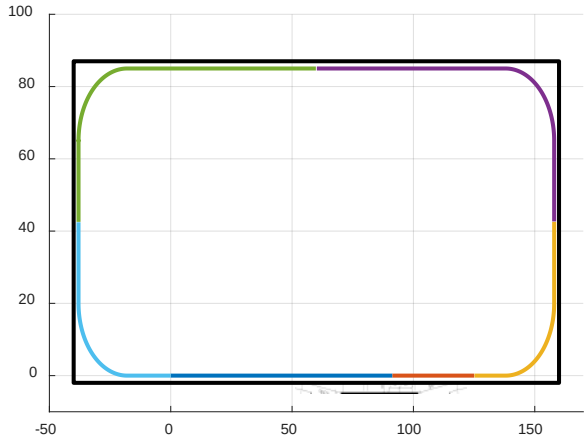


Figure 4.9: Centreline of path divided into 6 sections each one shown in different colour.

In Table 4.1 the elapsed time for each sector is shown with a total computing time of 9.65 seconds, resulting in a difference of 4.7 seconds.

Sector	*1*	*2*	*3*	*4*	*5*	*6*	*Total*
Time (s)	1.61	1.58	1.03	1.09	1.14	3.2	9.65

Table 4.1: Elapsed time per sector in solving the OCP (6 sectors).

- Dividing the path in 3 sectors.

Except for the bus stop section, the rest of the path geometry constrains are constant. A small test was carried out with the path divided in 3 sectors which consist in the bus stop, take-off and the rest of the path. The number of samples for the first 2 sections has been N=200 and the other one N=800. The computing time has been of 11.64 seconds, which is 2 seconds slower than the 6 sectors method and 2.71 seconds faster than computing the path planning problem at once. Hence, it might be confirmed that the more sections in which the path is divided, the faster the algorithm will find a global solution.

Sector	*1*	*2*	*3-6*	*Total*
Time (s)	1.61	1.58	8.45	11.64

Table 4.2: Elapsed time in solving the OCP for the path divided in 3 sectors.

4.1.3 Time optimal trajectory formulation comparison

In the following, the effect of the time optimal formulation explained in section 3.3.4 is shown. Taking again the basic scenario of the left side bus stop, a trayectory was found for different values of ρ:

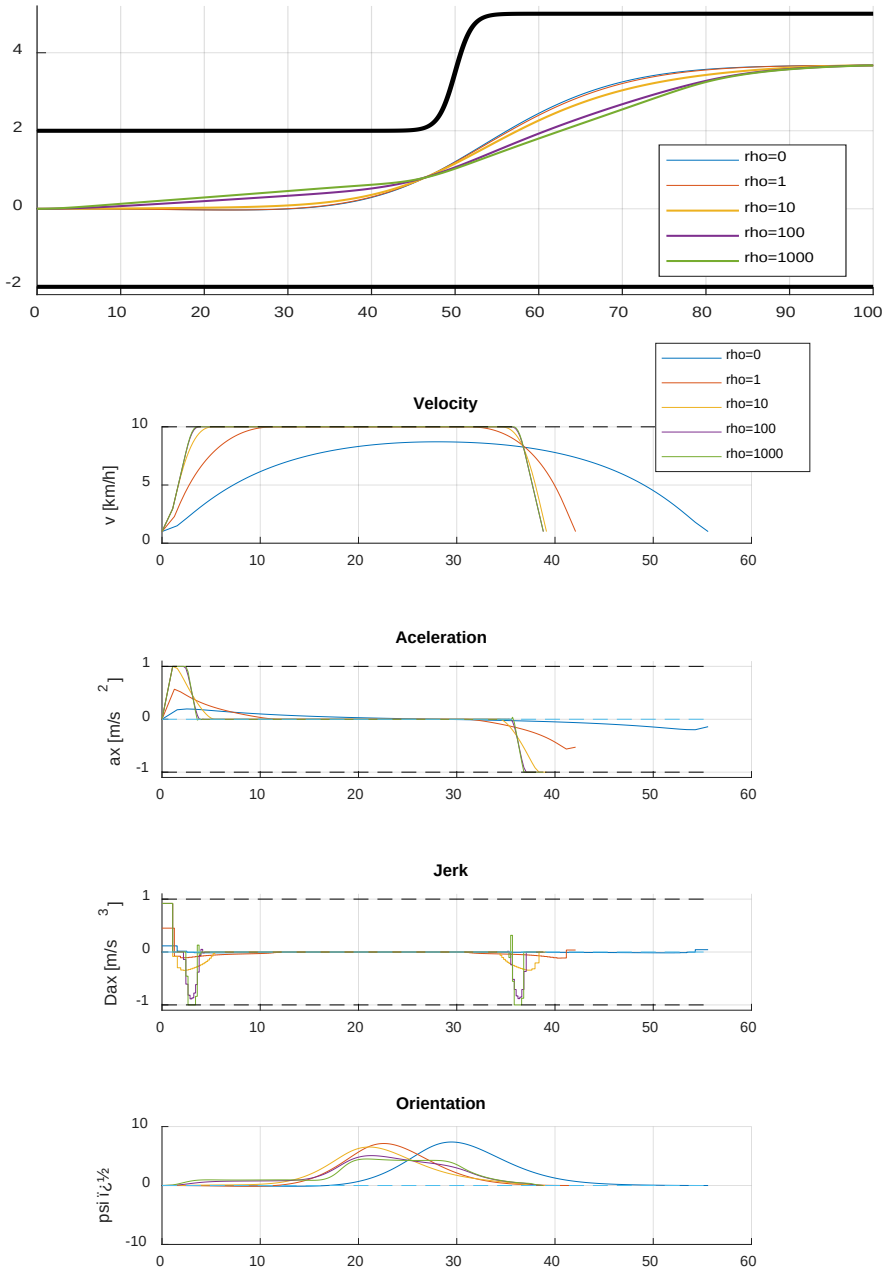


Figure 4.10: Time optimal formulation solution for different values of ρ.

In the previous figure, it can be seen how as the value of ρ increases, the bus arrives faster to the end, as the velocity reference (10 km/h) is reached earlier, starting at 1km/h. From values of $\rho \geq 10$ there is not a significative difference in the final time, but also for the values of $\rho = 100$,1000 the value of jerk is increased undesirable but inside the comfort limits. In addition, for $\rho = 10$ the path shown in the first plot does not differ too

much from the non-time optimal path, meeting with the comfort constrains, and it arrives 15.88 seconds faster without having as much jerk as for $\rho = 100$.

Another approach was mentioned in 3.3.4 where it was suggested the use of an unreachable velocity reference with a high penalty on the velocity instead of implementing the time optimal formulation in the cost function. Next figure shows a comparison between both methods:

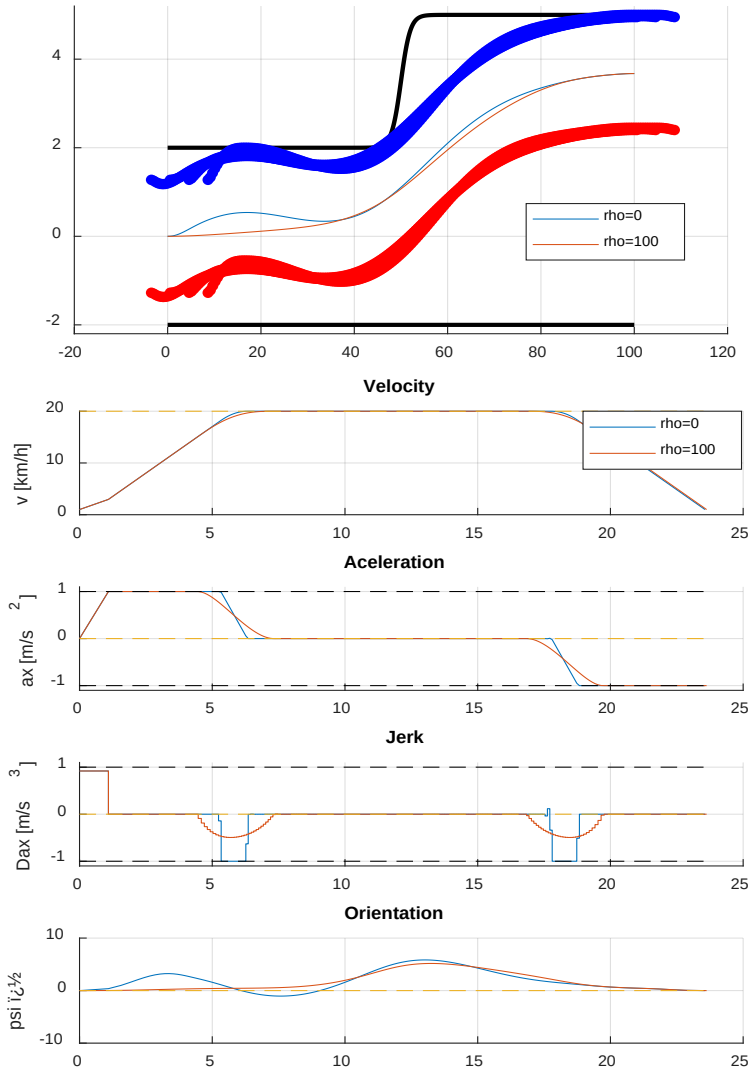


Figure 4.11: Time optimal methods comparison.

Both trajectories are completed in the same time but differences in the orientation and jerk are found, having a smoother trajectory when adding the time constraint to the cost function. In addition, using the time constraint in the cost function showed a decrease in the time needed to solve the OCP. For the method with a high penalty on velocity, the solver did its job in 4.05 seconds while the time penalty in the cost function took 1.6 seconds to solve the problem. For the closed path solutions, the use of the time constraint in the cost function helped the solver to find the solution when solving the whole path at once, while without it the solver was not able to find a solution.

4.1.4 Why a reference trajectory?

The OCP is solved using a cost function which penalises the deviations from a reference trajectory which is previously known. Hence, the closer the reference trajectory is to the optimal solution, less time will take for the NLP solver to converge into a solution. This means that for defined scenarios, the OCP can be solved using a standard reference but if the OCP wants to be solved later, for example, with different initial conditions, weights, or slightly different constraints, the solution that was found initially can be the new reference and the solver will converge into a solution faster (i.e. the solution of easier problems like just following a reference, can help to solve more complex problems faster). For example, firstly consider the case of a left side bus stop. The problem was solved 5 times and the solution of each iteration was the reference for the next iteration. The elapsed time for each iteration is shown in the following table, where it can be seen how the time for the second and third iterations is reduced, but remains the same for the last two iterations:

Iteration	*1*	*2*	*3*	*4*	*5*
Time (s)	1.5	1.13	1.08	1.09	1.08

Table 4.3: Elapsed time for each iteration, having as reference for iteration k, the solution of iteration (k-1).

Finally, for the more complex problem of the closed path trajectory where the solver needed 14.35 seconds in finding the solution. If the solution that has been found is introduced to the solver as a new reference trajectory, the time decreases significantly, around 2 seconds less to find the solution.

4.2 Results in Gazebo simulator

The trajectory solutions obtained for the right-side bus stop and circular path in sections 4.1.1 and 4.1.2 are wanted to be tested in the simulator to see how the path follower performs when a computer-generated trajectory is sent to it. This path follower, which is included in the Traffic Situation Management block from the architecture shown in Figure 3.19, is adapted and connected with Gazebo simulator as explained in section 3.4.3. However, and as it was mentioned in that section, this step of connecting Gazebo to the ROS framework, is not completed yet, resulting in the bus being unable to reach the goal after losing the trajectory. A comparison of the trajectory that has been sent (in an .xml file) to ROS and the actual trajectory that the bus model has followed is shown in the following figure, where it can be seen how the cannot follow the path correctly. Several causes can be behind this behaviour, being the wrong connection of the TSM to Gazebo or a bad tuning of the path follower controllers the most probable ones.

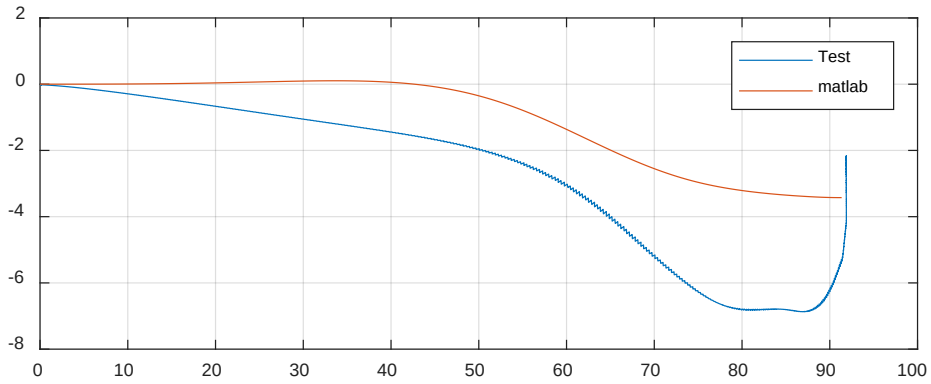


Figure 4.12: Trajectory comparison between MATLAB and Gazebo simulation.

4.3 Real test results

A real test is carried out in a squared area where the geometry of the bus stop shown in Figure 3.7 is built. By knowing the dimensions of the bus stop and its surroundings, the problems of a bus stop docking and a circular path driving can be solved in MATLAB with the actual measures to obtain a trajectory that is adapted to the format specified in section 3.4.4 so the ROS framework and the VTM block of Figure 3.19 can read it and send the necessary steering angle and velocity request to the bus. The distances from the initial point to the bus stop of the right-side bus stop MATLAB solution shown in section 4.1.1 are modified according to the dimensions of the area where the test takes place:

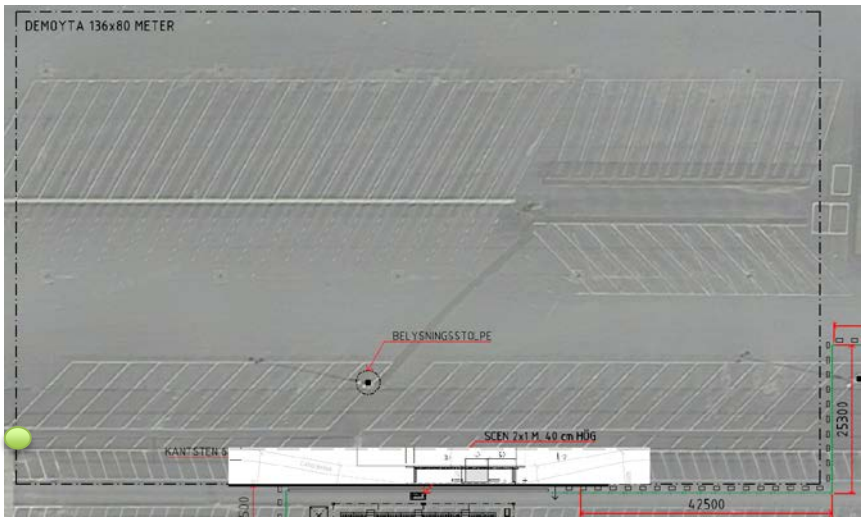


Figure 4.13: Area for the test with the real bus. Bus stop is delimited by the white rectangle. Green dot is the starting point.

The values of the weights from matrices Q, R are maintained. The initial speed is set to be $v_0 = 1\ km/h$ and the final speed this time is set to a closer value to 0, being finally of $v_f = 0.01\ km/h$, avoiding the singularity of the spatial space transformation. Maximum velocity is constrained to $v_{max} = 15\ km/h$ and the comfort constraints are

kept at $a_x \epsilon [-1,1]\frac{m}{s^2}$; $b \epsilon [-1,1]\frac{m}{s^3}$ and $a_y \epsilon [-1,1]\frac{m}{s^2}$. After having generated the trajectory in MATLAB, the offset for the initial real position and orientation $(X_{r0}, Y_{r0}, \psi_{r0})$ of the bus in its coordinate system has to be added to the trajectory. For that, the following matrix transform is applied:

$$T = \begin{pmatrix} \cos(\psi_{r0}) & -\sin(\psi_{r0}) & 0 & X_{r0} \\ \sin(\psi_{r0}) & \cos(\psi_{r0}) & 0 & Yr_0 \\ 0 & 0 & 1 & Z_{r0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

resulting the new values for the position and heading angle in the bus coordinate system:

$$(X_r, Y_r, Z_r, 1)^T = T \cdot (X, Y, Z, 1)^T \tag{4.3}$$

$$\psi_r = \psi - \psi_{r0} \tag{4.4}$$

The heading offset ψ_{r0} was obtained by aligning the bus with the real sidewalk to add the angle that the bus stop had. The trajectory is then executed and followed by the bus. showing a good tracking of the generated path with a slight deviation at the end. The results of the followed path and speed, acceleration, jerk, lateral acceleration and orientation is compared respect to the MATLAB trajectory in the following figure:

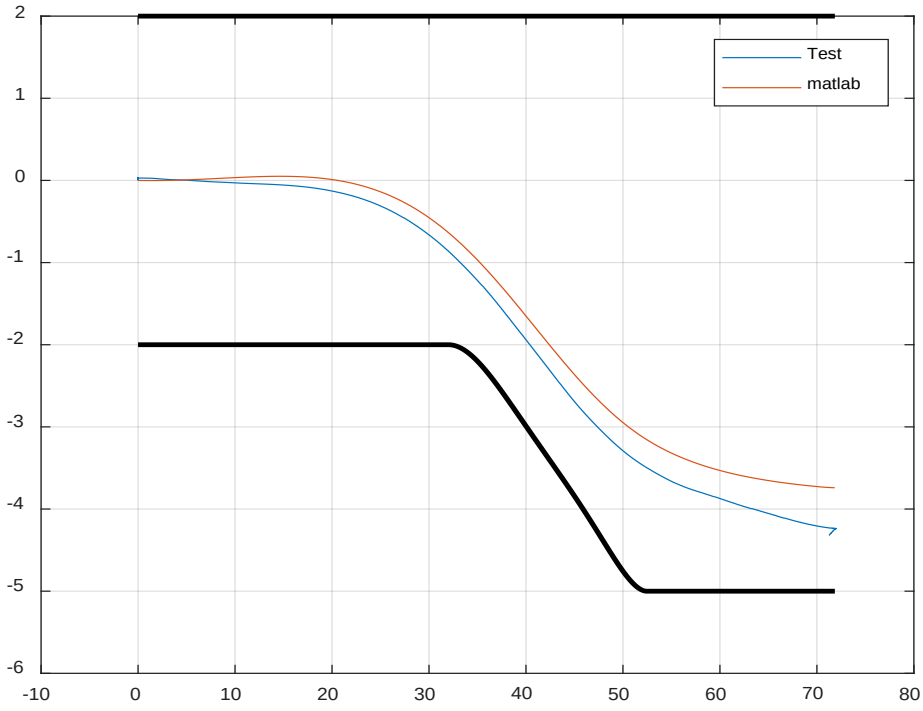


Figure 4.14: Path comparison between the computer-generated path and the path done by the autonomous bus.

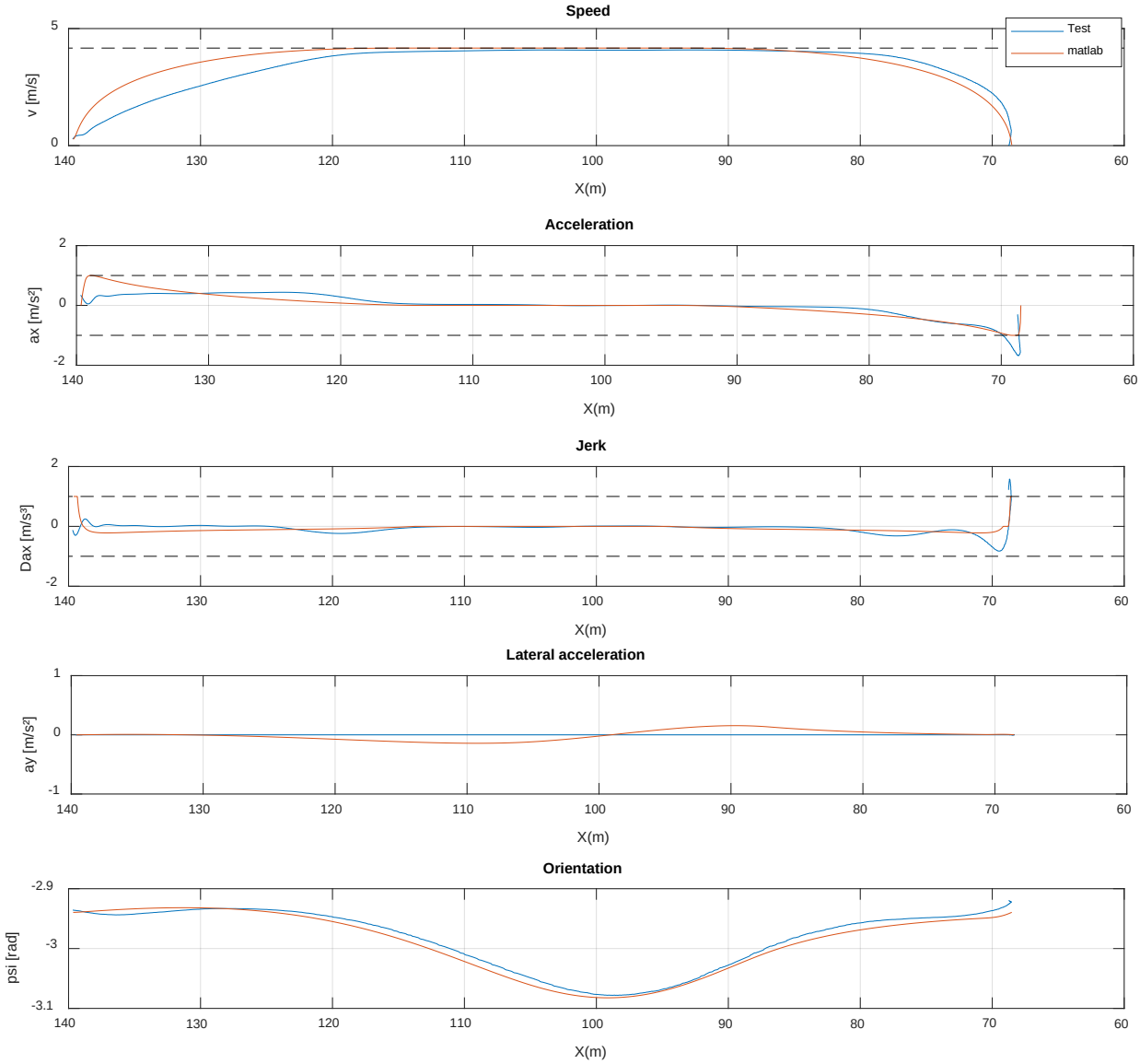


Figure 4.15: Comparison of the MATLAB trajectory and the trajectory that the bus actually did.

The trajectory followed by the bus shown in Figure 4.15 compared to the computer generated trajectory shows how the signals are very similar each other if plotted respect space. The speed constraint is met along the entire trajectory and the lateral acceleration too but, for the longitudinal acceleration, and jerk it can be seen that their values are inside the desired interval until the end, where the longitudinal acceleration goes beyond 1 m/s^2, concretely 1.65 m/s^2 and in consequence, the jerk goes over 1 m/s^3, violating the imposed constraints. Although it is not clear enough due to the early stages of the autonomous bus used, this increment in the acceleration when braking could be due to the need of a better adjustment in the vehicle controllers when reaching low velocities. On the other side, if the plots are made respect time, it can be seen how the time needed

to do the bus stop is 2.56 seconds slower than the theoretical trajectory followed by the bicycle kinematic model. This is acceptable due to the high mass of the real bus, along with the powertrain delays and transients that make the bus to take-off and brake slower. The trajectory ends 50 cm below the MATLAB trajectory, where the bus should have hit the sidewall, but again it did not. This suggest that the actual measures of the bus stop are not accurate and equal to the ones initially provided. Also, the position retrieved by the GPS and odometry from the bus has some tolerance that contributes to the lateral displacement error showed in addition with the way of measuring the initial heading offset, which rotates the whole trajectory computed in MATLAB and if it is not set accurately it introduces more displacement error. Finally, the passengers were satisfied in comfort aspects with the trajectory performed, including the braking, resulting in a smoother driving than in other tests which were carried out with manually driving recorded trajectories.

Finally, the trajectory for a closed path was tested. The bus had to take off, then go inside the bus stop and come back following a circled path:

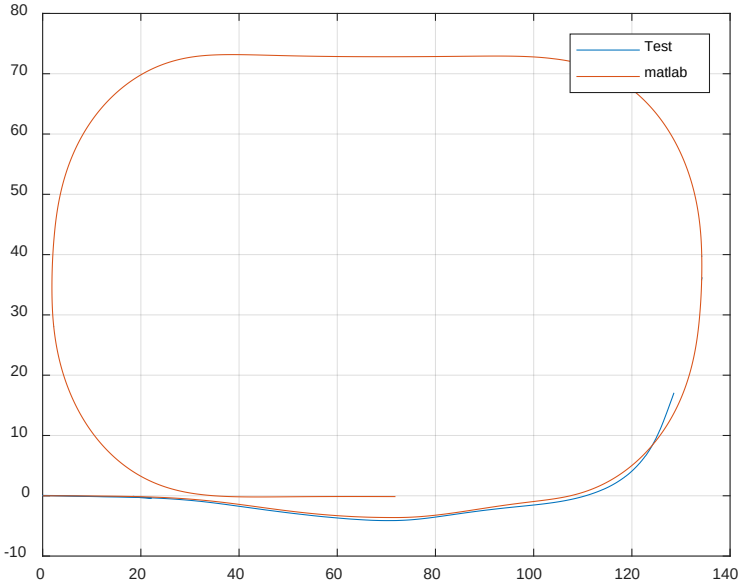


Figure 4.16: Comparison of the MATLAB circular trajectory and the trajectory followed by the bus. The path follower failed in the middle of the first turn.

Figure 4.16 shows the trajectory followed by the bus. The bus performed well when taking off, going inside and outside the bus stop with the desired speeds, however the path follower failed in the middle of the first turn, missing the trajectory and steering and accelerating hardly trying to find it again. In that moment, the bus driver took control of the bus for safety reasons. The reason of this behaviour in the middle of the first turn is not clear yet, but like the braking tuning problems mentioned above, it could be that a better tuning of the path follower had to be done. However, this are the first steps of a large project and this kind of failures are expected in these early stages. Performing a bus stop from a computer-generated trajectory is just the beginning of what is about to come.

5
Discussion

5.1 Conclusions

In this thesis project, a path planning algorithm has been developed, obtaining a trajectory that allows an autonomous bus to reach a goal, meeting lane geometry constraints to not go beyond the road limits, and meeting comfort constraints like maximum and minimum values of longitudinal and lateral accelerations, longitudinal jerk and speed. This trajectory has been obtained as a solution of an Optimal Control Problem solved using the multiple-shooting algorithm along with a symbolic tool software for optimization and Nonlinear Programs solving. For an efficient treatment of the boundaries and constraints of the lane so they are independent of the vehicle speed and easier to declare, a spatial model transformation has been proposed to formulate the OCP. Also, to improve the performance of the NLP solver, giving a good initial guess and reference helps to speed up the convergence. One way of doing this is solving a simpler OCP problem and send this solution as an initial guess for more complex OCP problems.

The use of a kinematic bicycle model has shown to be a good choice for a bus path planning algorithm due to low speeds and lack of slip effects and its simplicity compared to the model of a real bus which allows the finding of an accurate trajectory in less time that can be followed after by a real model of the bus with a good path follower resulting in a similar trajectory.

This algorithm has been tested for different scenarios, which are the case of a bus stop, a take-off and a circular path driving. For the first scenario, the algorithm was modified to work in several bus stops geometries which are commonly found in cities, obtaining optimal trajectories which are subject to the constrains mentioned above. By modifying the constrains of the problem, the algorithm worked well for the other two scenarios. Additionally, a time optimal trajectory formulation was added to the algorithm improving the results returned from the solver.

Finally, the computer-generated trajectories were tested on the real autonomous bus for the bus stop docking scenario and a circular path driving. The results showed that, in spite of the transients introduced by the real bus model dynamics, the path is followed as expected, resulting in a followed trajectory similar to the computer-generated one. Some constraints are only violated at the end when braking due to the need of better tuning of the controllers of the real bus. The passengers found the driving for the bus stop as fully acceptable. For a circular path, the path follower failed, losing the trajectory to be followed and having to abort the manoeuvre.

5.2 Future work

The work that has been done in this thesis is part of a much larger project as mentioned in the introduction. Therefore, several aspects can be improved of the algorithm presented in this thesis:

- Improve the computation time of the algorithm to be faster and be able to use it not only as a global planner, but also as a local planner in real time. In this way, optimal global trajectories can be computed as now, but local trajectories could be also computed in order to avoid unexpected obstacles or changes in the boundaries of the road. These local trajectories can be also the result of an OCP, minimizing the already mentioned constraints.

- CasADi allows to export the code from other languages to C++. Implementing this algorithm in C++ instead of in MATLAB will not only speed up the computation time, but also it is possible to create a standalone ROS node that is being executed with a certain frequency, allowing its usage to recompute paths in a faster way.

- The use of a more complex vehicle model equations when solving the OCP will result in computer-generated trajectories that will be faithful to the actual trajectories that the bus did when trying to follow a trajectory generated with the kinematic bicycle model.

- Add exceptions for the singularity problems when reaching velocities of 0 km/h. Although good results were obtained in the real bus with final velocities of 0.01 km/h or 1 km/h, it is a good idea to add this capability to improve the flexibility of the algorithm.

- The fully integration of the generated trajectory with the simulator architecture based on ROS and Gazebo in order to speed up the testing of the future algorithms with a high-fidelity model of the bus.

Bibliography

[1] A. Nikitas, I. Kougias, E. Alyavina and E. N. Tchouamou, “How can autonomous and connected vehicles, Electromobility, BRT, Hyperloop, Shared Use Mobility and Mobility-as-a-Service Shape Transport Futures for the context of smart cities?,” Urban Science. MDPI, 2017.

[2] A. Lajuen, “Energy consumption and cost-benefit analysis of hybrid and electic city buses.,” in *Transportation Research Part C: Emerging technologies.*, Elsevier, 2016, pp. 1-15.

[3] A. Lajunen and T. Lipman, “Lifecycle cost assessment and carbon dioxide emissions of diesel, natural gas, hybrid electric, fuel cell hybrid and electric transit buses,” Energy, 2016, pp. 329-342.

[4] A. Talebpour and H. Mahmassani, “Influence of connected and autonomous vehicles on traffic flow stability and throughput.,” in *Transportation Research Part C: Emerging Technologies*, vol. 71, Elsevier, 2016, pp. 143-163.

[5] N. J. Patel, “Ride Comfort Evaluation for Autonomous City Buses,” Universität Stuttgart, Stuttgart, 2018.

[6] R. Verschueren, S. De Bruyne, M. Zanon, J. V. Frasch and M. Diehl, “Towards Time-Optimal Race Car Driving using nonlinear MPC in Real-Time,” in *Proceedings of the 53rd Conference on Decision and Control*, 2016.

[7] R. Verschueren, M. Zanon, R. Quirynen and M. Diehl, “Time-optimal Race Car Driving using an Online Exact Hessian based Nonlinear MPC Algorithm,” in *Proceedings of the European Control Conference*, 2016.

[8] Y. Gao, A. Gray, J. V. Frasch, T. Lin, E. Tseng, J. K. Hedrick and F. Borrelli, “Spatial predictive control for agile semi-autonomous ground vehicles,” in *Proceedings of the 11th International symposium on Advanced Vehicle Control*, 2012.

[9] J. V. Frasch, A. Gray, M. Zanon, H. J. Ferreau, S. Sager, F. Borrelli and M. Diehl, “An Auto-generated Nonlinear MPC algorithm for Real Time Obstacle Avoidance of Ground Vehicles,” in *Proceedings of the European Control Conference*, 2013.

[10] M. Diehl, H. G. Bock, H. Diedam and P. B. Wieber, “Fast Direct Multiple Shooting Algorithms for Optimal Robot Control,” HAL, Heidelberg, Germany, 2005.

[11] R. Bellman, Dynamic Programming, University Press, Pinceton, 1957.

[12] H. Bock and K. Plitt, “A multiple shooting algorithm for direct solution of optimal control problems,” in *Proceedings 9th IFAC World Congress Budapest*, 1984.

[13] J. Andersson, "A General-Purpose Software Framework for Dynamic Optimization", PhD thesis, Arenberg Doctoral School, KU Leuven, 2013.

[14] J. Andersson, J. Gillis and M. Diehl, “User Documentation for CasADi v3.3.0,” Github, February 11, 2018.

[15] A. Verheyleweghen and C. Backi, “Virtualsimlab,” 27 October 2016. [Online]. Available: http://www.virtualsimlab.com/archive/. [Accessed 14 06 2018].

[16] A. Wätcher, “An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering,” Ph.D. dissertation Carnegie Mellon University, 2002.

[17] Sektion Utformning av vägar och gator, Vägar och gators utformning, VGU, Stockholm: Svenska Kommunförbundet, 2004-05.

[18] Mathworks, [Online]. Available: https://www.mathworks.com/help/optim/ug/lsqcurvefit.html#References. [Accessed 18 06 18].

[19] ROS, [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Accessed 18 06 2018].

[20] “Gazebo,” [Online]. Available: http://gazebosim.org/tutorials. [Accessed 18 06 2018].

[21] T. Victor, M. Rothoff, E. Coelingh, A. Ödblom and K. Burgdorf, “When autonomous vehicles are introduced on a larger scale in the road transport system: The Drive Me project.,” *Automated Driving*, pp. 541-546, 2016.

[22] SAE International, “SAE,” [Online]. Available: http://www.sae.org/misc/pdfs/automated_driving.pdf. [Accessed 15 June 2018].

[23] J. Eriksson and L. Svensson, “Tuning for Ride Quality in Autonomous Vehicle. Application to Linear Quadratic Path Planning Algorithm,” Uppsala Universitet, Uppsala, June, 2015.

[24] N. Loulloudes, 4 March 2015. [Online]. Available: http://www.cs.ucy.ac.cy/~nickl/talks/V2XUnivNicosia.pdf. [Accessed 15 June 2018].

[25] K. Kottenhoff, Driving styles and the effect on passengers. Developing comfort indicators., Stockholm: KTH, 2016.

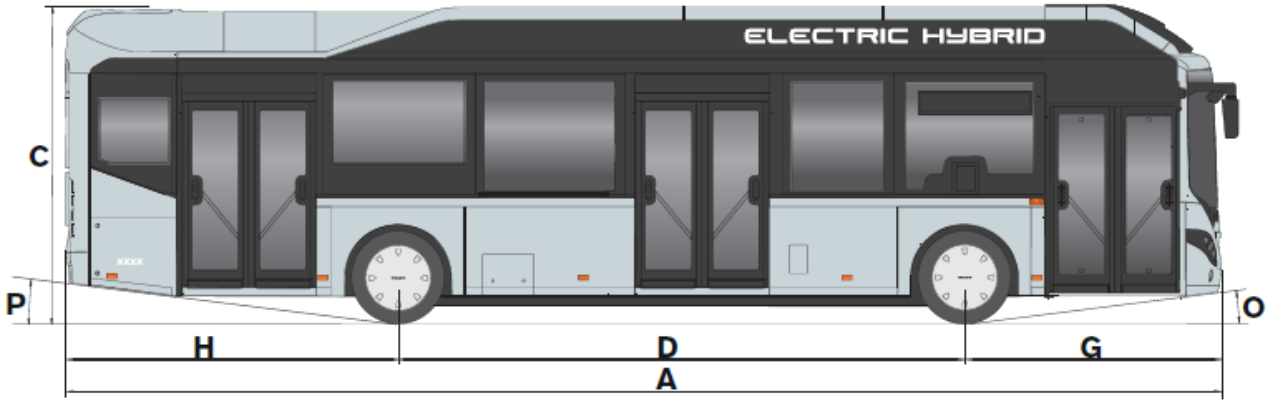
Appendices

A. Volvo 7900 Electric Hybrid Bus

Volvo Buses. Driving quality of life

VOLVO 7900 ELECTRIC HYBRID

Euro 6



Model	4 x 2, 10.6 m

Overall dimensions		
A	Overall length (mm)	12134
	Overall width (mm)	2550
C	Overall height (mm)	3300
D	Wheelbase (mm)	5945
G	Front overhang (mm)	2704
H	Rear overhang (mm)	3485
O	Approach angle (°)	6.1
P	Departure angle (°)	6.0
	Turning radius (outer front corner) (mm)	10910
	Turning radius (outer front wheel) (mm)	8970
	Overall height, approach and departure angles with tyres	275/70 R22.5

B. MATLAB code

In this appendix, the code of the circular path is included as an example that involves the case of a bus stop docking, a take-off, and the modifications for the local curvatures and time optimal solutions. It also can be seen how the different constraints have been implemented along the path and how the results can be plotted.

```
clear all
close all
clc


addpath('/home/alex/Escritorio/TFM/MATLAB/casadi-matlabR2014b-v3.2.3')
import casadi.*


centreline=load('centreline'); %Load data of centreline
Sf = centreline.distance;  % Total path distance
N = 1000;   % Number of grid points
ts = Sf/N; % sampling space (ss)
n_int_steps = 5;


x0 = [0,1/3.6,0,0,0,0]; %Initial conditions


v_min=1;    % Minimum velocity
v_max=30;   % Maximum velocity
v_ref=30;   % Velocity reference (constant)
v_end=1;    % Final velocity


% Bus parameters
L = 5.945;      % D  distance between wheel axels
Lb = -3.485;    % H  rear part from rear wheel
Lf = L + 2.704; % G  front part starting from front wheel
width = 2.55/2; % Width of rear axle from midpoint
n_points   = 4; % number of points used to discretise the bus side
point_grid = linspace(Lb,Lf,n_points);


Daxmax = 1;      % maximum longitudinal jerk
axmax  = 1;      % maximum longitudinal acceleration
aymax  = 1;      % maximum lateral acceleration


% Load geometry of bus stop
ySS=load('busgeo1');
PP1=ySS.PP';
ySS1=load('busgeo1ref.mat');
PPref1=ySS1.PP';


ySS=load('busgeo2');
PP2=ySS.PP';
ySS1=load('busgeo2ref.mat');
PPref2=ySS1.PP';


% Cost tuning of OCP
% Q = diag([1e-2,1e-2,1e-0,1,1,0]);
Q = diag([0.01,0.1,1,1,1,0]);
R = diag([1e-0,10]);
W = blkdiag(Q,R); % Concatenates matrices in diagonal
P = Q;


% Define states and controls
xLabels = {'e_y','v','a','e_psi','delta','t'}; % Lateral displacement, v, a
                                               % angular deviation,
                                               % steering angle
uLabels = {'Da','Ddelta'}; % Derivative of a (jerk) and w (steering rate)
```

```
nx = numel(xLabels);
nu = numel(uLabels);
for k = 1:nx
    xI.(char(xLabels{k})) = k;  % Structure to store states values 1 to 6
end


for k = 1:nu
    uI.(char(uLabels{k})) = k;  % Structure to store control values 1 to 6
end
xuI.x = xI;   % Structure of structures
xuI.u = uI;


% Generate symbolic casadi variables
% States and controls
x  = SX.sym('x',[numel(xLabels),1]);
u  = SX.sym('u',[numel(uLabels),1]);
% Reference
xr = SX.sym('xr',[numel(xLabels),1]);
ur = SX.sym('ur',[numel(uLabels),1]);
kappa = SX.sym('kappa',1);


% Define the right hand side of the ODE and lump into a casadi function
% kappa = 0; % K sigma: local curvature of the curve sigma
sdot = 1/(1-kappa*x(xI.e_y)) * x(xI.v)*cos(x(xI.e_psi));


psidot = x(xI.v)*tan(x(xI.delta))/L; % v/L * tan(delta)
rhs = [ x(xI.v)*sin(x(xI.e_psi)) / sdot;
        x(xI.a) / sdot;
        u(uI.Da) / sdot;
        psidot / sdot - kappa;
        u(uI.Ddelta) / sdot;
        1 / sdot;      ];


% Function(name, SX arg, SX res, argn, resn)
f = Function('f',{x,u,kappa},{rhs});


% Lateral acceleration
ay = (x(xI.v))^2*tan(x(xI.delta))/L;
out = Function('out',{x,u},{ay});


dt = ts/n_int_steps;


% Define an integrator Runge Kutta order 4 and put into a function
[k1] = f(x,u,kappa);
[k2] = f(x+0.5*dt*k1,u,kappa);
[k3] = f(x+0.5*dt*k2,u,kappa);
[k4] = f(x+dt*k3,u,kappa);
rk4_step = Function('rk4_step',{x,u,kappa},...
    {x + (1./6)*dt*(k1 + 2*k2 + 2*k3 + k4)});


x_ = x;
for k = 1:n_int_steps
    x_ = rk4_step(x_,u,kappa);
end
rk4 = Function('rk4',{x,u,kappa},{x_});
rk4_x = rk4.jacobian(0);
rk4_u = rk4.jacobian(1);


% Define functions for the stage and terminal cost
xu = [x;u];
xur = [xr;ur];
ell  = Function('ell',{x,u,xr,ur},...
    { (xu-xur).'*W*(xu-xur) },[xLabels,uLabels,xLabels,uLabels],{'ell'});
ellF = Function('ellF',{x,xr},...
    { (x-xr).'*P*(x-xr) },[xLabels,uLabels,xLabels,uLabels],{'ellF'});



% Create OCP variables and parameter variables
```

```
V  = MX.sym('V',  N*nu + (N+1)*nx );
Vr = MX.sym('Vr', N*nu + (N+1)*nx );
rho = MX.sym('rho', 1 ); % rho constant for time optimality
KAPPA = MX.sym('KAPPA',N+1); % Local curvatures


offset = 0;
for k = 1:N
    for j = 1:nx
        iV_(k).x.(char(xLabels{j})) = j + offset;
    end
    offset = offset + nx;
    for j = 1:nu
        iV_(k).u.(char(uLabels{j})) = j + offset;
    end
    offset = offset + nu;
end
for j = 1:nx
    iV_(N+1).x.(char(xLabels{j})) = j + offset;
end
for j = 1:nu
    iV_(N+1).u.(char(uLabels{j})) = zeros(0,1);
end


% Define indexing functions for the optimisation and parameter variables
iV1 = @(fieldname,interval) reshape(cell2mat(arrayfun(@(x)
struct2array(x.(fieldname)),iV_(interval),'UniformOutput',false)).',[],1);
iV2 = @(fieldname,interval,secondname) reshape(cellfun(@(y)
y(xuI.(fieldname).(secondname)), arrayfun(@(x)
struct2array(x.(fieldname)),iV_(interval),'UniformOutput',false)),[],1);
iif = @(varargin) varargin{2 * find([varargin{1:2:end}], 1, 'first')}();
iV = @(varargin) iif( nargin == 1, @() iV1(varargin{1},':'), ...
                      nargin == 2, @() iV1(varargin{1:2}), ...
                      nargin == 3, @() iV2(varargin{1:3})...
                      );
%%
% Construct de OCP by declaring the cost function and constrains
cost = 0;
g = []; % Vector of constrains
v_lb=[];
iG_.dyn = {};
iG_.ay = {};
offset = 0;


for k = 1:N
    X_ = rk4( V(iV('x',k)), V(iV('u',k)),KAPPA(k)); % Use the integrator to compute
where the state will end up if I use that control
    g = [g; V(iV('x',k+1)) - X_]; % x_k+1 = f(x_k,u_k)
    iG_.dyn{k} = (1:nx).' + offset;
    offset = offset + nx;


    ayk = out(V(iV('x',k)), V(iV('u',k))); % Evaluate lateral acceleration and add it to
vector of constrains
    g = [g; ayk];
    iG_.ay{k} = offset + 1;
    offset = offset + 1;


    if k > 1
        % position along the reference: s_k = k * ts
        psip = V(iV('x',k,'e_psi')); % Need to add psi_curve if different form 0


        iG_.y_max{k} = [];
        iG_.y_min{k} = [];
        for j = 1:n_points


            xp = point_grid(j)*cos( psip ) - width*sin( psip );
            dyp = point_grid(j)*sin( psip ) + width*cos( psip );
            yp = V(iV('x',k,'e_y')) + dyp;
            e_y_max_k = lane_width/2; %Upper geometry constrain
            g = [g; yp - e_y_max_k]; % The point - the maximum deviation
            iG_.y_max{k} = [iG_.y_max{k}; 1 + offset];
```

```
            offset = offset + 1;

            dyp = point_grid(j)*sin( psip ) - width*cos( psip );
            yp = V(iV('x',k,'e_y')) + dyp;

            %Lower geometry constrain
            xx=(k-1)*ts+xp;
            e_y_min_k=if_else(xx<51.5,-2,if_else(xx>=51.5 &
xx<=72,polyval(PP1',xx),if_else(xx>72 & xx<101.5,-5,...
                if_else(xx>=101.5 & xx<=122,polyval(PP2',xx),-2))));

            g = [g; yp - e_y_min_k];
            iG_.y_min{k} = [iG_.y_min{k}; 1 + offset];
            offset = offset + 1;
        end

    end

cost = cost + ell( V(iV('x',k)), V(iV('u',k)), Vr(iV('x',k)), Vr(iV('u',k)) ); % Cost
end
%%
cost = cost + ellF( V(iV('x',N+1)), Vr(iV('x',N+1)) ); % Final cost
cost = cost + rho*V(iV('x',N+1,'t')); % Time cost



ng = size(g);
iG1 = @(fieldname,interval) vertcat(iG_.(fieldname){interval});
iG = @(varargin) iif( nargin == 1, @() iG1(varargin{1},':'), ...
                      nargin == 2, @() iG1(varargin{1:2}));

% Declare the NLP function and pass it to a NLP solver
nlp = struct( 'x', V, 'p', [Vr;rho;KAPPA], 'f', cost, 'g', g); %x is optimization
variable, p known parameter vector  g constrains
nlpFun = Function('nlpFun',{V,[Vr;rho;KAPPA]},{cost,g});

opts = struct('ipopt',struct());
solver = nlpsol( 'solver', 'ipopt', nlp, opts );
% CASADI % Compute all derivatives to send to the solver
% nlpsol('solver','ipopt'(for nlp), structur,,,,)

% Define upper and lower bounds for constrains. Equality constrains
% are implemented by setting the lower equal to upper bound
%%
lbv = -inf*ones(size(V));
ubv =  inf*ones(size(V));
lbg = -inf*ones(ng);
ubg =  inf*ones(ng);

% Bounds are separated for efficiency.
% Bounds of constrains
lbg(iG('dyn')) = zeros( size(iG('dyn')) );
ubg(iG('dyn')) = zeros( size(iG('dyn')) );
% lbg(iG('y')) = ones( size(iG('y')) );
ubg(iG('y_max')) = zeros( size(iG('y_max')) );
lbg(iG('y_min')) = zeros( size(iG('y_min')) );

ubg(iG('ay')) =  aymax*ones( size(iG('ay')) );
lbg(iG('ay')) = -aymax*ones( size(iG('ay')) );

% Bounds of variables
lbv(iV('u',1:N,'Da')) = -Daxmax*ones(size(iV('u',1:N,'Da')));
ubv(iV('u',1:N,'Da')) =  Daxmax*ones(size(iV('u',1:N,'Da')));

lbv(iV('x',':','a')) = -axmax*ones(size(iV('x',':','a')));
ubv(iV('x',':','a')) =  axmax*ones(size(iV('x',':','a')));
```

```
lbv(iV('x',':','v')) = zeros(size(iV('x',':','v'))) + v_min/3.6;
ubv(iV('x',':','v')) = ones(size(iV('x',':','v')))*v_max/3.6;
lbv(iV('x',floor(centreline.distances(1)/ts),'v'))=v_end/3.6;
ubv(iV('x',floor(centreline.distances(1)/ts),'v'))=v_end/3.6;
ubv(iV('x',N+1,'v')) = v_end/3.6;
lbv(iV('x',floor(centreline.distances(1)/ts),'e_y'))=-5+width+0.2;
ubv(iV('x',floor(centreline.distances(1)/ts),'e_y'))=-5+width+0.2;


lbv(iV('x',floor(centreline.distances(1)/ts),'e_psi'))=0;
ubv(iV('x',floor(centreline.distances(1)/ts),'e_psi'))=0;


lbv(iV('x',floor(centreline.distances(1)/ts),'delta'))=0;
ubv(iV('x',floor(centreline.distances(1)/ts),'delta'))=0;


lbv(iV('x',N+1,'e_y')) = 0;
ubv(iV('x',N+1,'e_y')) = 0;


lbv(iV('x',N+1,'e_psi')) = 0;
ubv(iV('x',N+1,'e_psi')) = 0;


lbv(iV('x',N+1,'delta')) = 0;
ubv(iV('x',N+1,'delta')) = 0;

% Declare the reference for the states and controls
Vref = zeros(size(V));
Vref(iV('x',':','v')) = Vref(iV('x',':','v')) + v_ref/3.6;


for k = 1:N+1
            xx=(k-1)*ts;
            if(xx>=51.5 && xx<=51.5+20.5)
                    Vref(iV('x',k,'e_y'))=polyval(PPref1',xx);
            elseif(xx>51.5+20.5 && xx<101.5)
                    Vref(iV('x',k,'e_y'))=-5+width;
            elseif(xx>=101.5 && xx<=122)
                    Vref(iV('x',k,'e_y'))=polyval(PPref2',xx);
            else
                    Vref(iV('x',k,'e_y'))=0;
            end
end


% Declare the initial guess to be passed to the optimiser
x_init = zeros(size(V));
x_init(iV('x',':','v')) = v_ref/3.6;


% Initial state declare the initial constrain.
lbv(iV('x',1)) = x0;
ubv(iV('x',1)) = x0;


% Create vector of local curvatures
kappa_ref=[];
for k=1:N+1
    pos=(k-1)*ts;
    if (pos>=(centreline.distances_cum(5)+centreline.y_6(1)-centreline.yp9(1)))&&...
            (pos<=(centreline.distances_cum(5)+centreline.y_6(1)-
centreline.yp9(1)+(2*pi*centreline.r)/4))
        kappa_ref=[kappa_ref 0.05];
    elseif (pos>=138 && pos<=169.4159)
        kappa_ref=[kappa_ref 0.05];
    elseif (pos>=centreline.distances_cum(3)+(centreline.yp5(1)-...
            centreline.y_4(1))&& pos<=centreline.distances_cum(3)+(centreline.yp5(1)-...
            centreline.y_4(1))+(2*pi*centreline.r)/4)
        kappa_ref=[kappa_ref 0.05];


    elseif (pos>=(centreline.distances_cum(4)+centreline.x_5(1)-centreline.xp7(1)))&&...
            (pos<=(centreline.distances_cum(4)+centreline.x_5(1)-
centreline.xp7(1)+(2*pi*centreline.r)/4))
        kappa_ref=[kappa_ref 0.05];
    else
```

```
        kappa_ref=[kappa_ref 0];
    end
end



tic
% call the solver and retrieve the solution
sol =
solver('x0',x_init,'lbx',lbv,'ubx',ubv,'lbg',lbg,'ubg',ubg,'p',[Vref;10;kappa_ref']);
v_opt = full(sol.x);
toc

% Extract final values of the solution and store them
e_x_end=centreline.distance;
e_y_end=v_opt(iV('x',':','e_y')); e_y_end=e_y_end(end);
v_end=v_opt(iV('x',':','v')); v_end=v_end(end);
a_end=v_opt(iV('x',':','a')); a_end=a_end(end);
e_psi_end=v_opt(iV('x',':','e_psi')); e_psi_end=e_psi_end(end);
delta_end=v_opt(iV('x',':','delta')); delta_end=delta_end(end);
t_end=v_opt(iV('x',':','t')); t_end=t_end(end);
save('IC16','e_y_end','v_end','a_end','e_psi_end','delta_end','t_end','e_x_end');



%% Plot results
[fnum,gnum]=nlpFun(v_opt,[Vref;10;kappa_ref']);
ay = full(gnum(iG('ay',1:N)));

spaceXstart=0;
spaceX = linspace(spaceXstart,N*ts+spaceXstart,N+1);
spaceU = linspace(spaceXstart,N*ts+spaceXstart,N);

timeX = v_opt(iV('x',':','t'));
timeU = timeX(1:end-1);
timeU2 = reshape([timeX,timeX].',1,[]);
timeU2 = timeU2(2:end-1);

figure(1)
%clf
Sf=100;
offset=Sf-48.5;
I=imread('stop.png');
image('CData',I,'Xdata',[0+offset 70.5+offset],'YData',[-2,-5])
hold on
plot([-40 160 160 -40 -40],[-2 -2 87 87 -2], 'k','LineWidth',2)
% plot(xS,yS,'m','Linewidth',2)
% plot(xS,xS*0+lane_width/2,'k','Linewidth',2)

axis([-45 165 -5 90])
x_16=[centreline.x_1,centreline.x_2,centreline.x_3,centreline.x_4,centreline.x_5,centrel
ine.x_6];
y_16=[centreline.y_1,centreline.y_2,centreline.y_3,centreline.y_4,centreline.y_5,centrel
ine.y_6];
resize_traj=interparc(N+1,x_16,y_16,'spline');
x_16=resize_traj(:,1)';
y_16=resize_traj(:,2)';
X_recov=[];
Y_recov=[];
PSI_recov=[];
firstindexcurve=find(kappa_ref==0.05,1,'first');
lastindexcurve=find(kappa_ref==0.05,1,'last');

% Improve this plotting
for k = 1:N+1
   if k<263
       X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(0))];
       Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(0))];
       PSI_recov=[PSI_recov 0+v_opt(iV('x',k,'e_psi'))];
   elseif k>=263 && k<=322
       inc=(pi/2)/(322-263);
```

```
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin((k-263)*inc))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos((k-263)*inc))];
        PSI_recov=[PSI_recov (k-263)*inc+v_opt(iV('x',k,'e_psi'))];
    elseif k>=323 && k<=382
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(pi/2))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(pi/2))];
        PSI_recov=[PSI_recov (pi/2)+v_opt(iV('x',k,'e_psi'))];
    elseif k>=383 && k<=441
        inc=(pi/2)/(441-383);
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(pi/2+(k-441)*inc))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(pi/2+(k-441)*inc))];
        PSI_recov=[PSI_recov (pi/2)+(k-383)*inc+v_opt(iV('x',k,'e_psi'))];
     elseif k>=442 && k<=689
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(pi))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(pi))];
        PSI_recov=[PSI_recov pi+v_opt(iV('x',k,'e_psi'))];
    elseif k>=690 && k<=748
        inc=(pi/2)/(748-690);
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(pi+(k-690)*inc))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(pi+(k-690)*inc))];
        PSI_recov=[PSI_recov pi+(k-690)*inc+v_opt(iV('x',k,'e_psi'))];
    elseif k>=749 && k<=786
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(3*pi/2))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(3*pi/2))];
        PSI_recov=[PSI_recov (3*pi/2)+v_opt(iV('x',k,'e_psi'))];
    elseif k>=787 && k<=845
        inc=(pi/2)/(845-787);
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin((3*pi/2)+(k-787)*inc))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos((3*pi/2)+(k-787)*inc))];
        PSI_recov=[PSI_recov (3*pi/2)+(k-787)*inc+v_opt(iV('x',k,'e_psi'))];
    else
        X_recov=[X_recov,(x_16(1,k)-v_opt(iV('x',k,'e_y'))*sin(2*pi))];
        Y_recov=[Y_recov,(y_16(1,k)+v_opt(iV('x',k,'e_y'))*cos(2*pi))];
        PSI_recov=[PSI_recov (2*pi)+v_opt(iV('x',k,'e_psi'))];
    end
end
plot(X_recov,Y_recov,'r');% Optimal path
x_36=[centreline.x_3,centreline.x_4,centreline.x_5,centreline.x_6]; % Reference after
bus stop
y_36=[centreline.y_3,centreline.y_4,centreline.y_5,centreline.y_6];
plot(spaceX(1,1:floor(126/ts)),Vref(iV('x',1:floor(126/ts),'e_y')),'b--')  % Reference
path
plot(x_36,y_36,'b--');%Reference path
grid on
% axis([-20 165 -10 10])



figure(2)
%clf
ax1 = subplot(7,1,1); % Controller velocity
hold on
plot(timeX,3.6*v_opt(iV('x',':','v')))
plot(timeX,3.6*Vref(iV('x',':','v')),'--')
title('Velocity');
ylabel('v [km/h]')
grid on
ax2 = subplot(7,1,2);
hold on
plot(timeX,v_opt(iV('x',':','a')))
plot(timeX,Vref(iV('x',':','a')),'--')
plot(timeX,timeX*0 + axmax,'--k')
plot(timeX,timeX*0 - axmax,'--k')
title('Aceleration');
ylabel('ax [m/s^2]')
ylim(1.1*[-axmax,axmax])
grid on
ax3 = subplot(7,1,3); % Jerk
hold on
Dax = v_opt(iV('u',1:N,'Da'));
Daxref = Vref(iV('u',1:N,'Da'));
plot(timeU2,reshape([Dax,Dax].',1,[]))
plot(timeU2,reshape([Daxref,Daxref].',1,[]),'--')
plot(timeX,timeX*0 + Daxmax,'--k')
```

```
plot(timeX,timeX*0 - Daxmax,'--k')
title('Jerk');
ylabel('Dax [m/s^3]')
ylim(1.1*[-Daxmax,Daxmax])
grid on
ax4 = subplot(7,1,4); % Angular deviation
hold on
plot(timeX,180/pi*v_opt(iV('x',':','e_psi')))
plot(timeX,Vref(iV('x',':','e_psi')),'--')
title('Orientation');
ylabel('psi')
grid on
% linkaxes([ax1,ax2,ax3,ax4],'x');
ax5 = subplot(7,1,5);
hold on
plot(timeX,180/pi*v_opt(iV('x',':','delta')))
plot(timeX,180/pi*Vref(iV('x',':','delta')),'--')
ylabel('delta')
title('Steering angle');
grid on
ax6 = subplot(7,1,6);
hold on
Ddelta = v_opt(iV('u',1:N,'Ddelta'));
Ddeltaref = Vref(iV('u',1:N,'Ddelta'));
plot(timeU2,reshape([Ddelta,Ddelta].',1,[]))
plot(timeU2,reshape([Ddeltaref,Ddeltaref].',1,[]),'--')
title('Steering rate w')
ax7 = subplot(7,1,7);
hold on
plot(timeU2,reshape([ay,ay].',1,[]))
% plot(timeU,ay)
plot(timeX,timeX*0 + aymax,'--k')
plot(timeX,timeX*0 - aymax,'--k')
ylabel('ay')
title('Lateral aceleration');
ylim(1.1*[-aymax,aymax])
grid on


% Write XML file with trajectory
route='/home/alex/Escritorio/TEST_1';
spaceX=X_recov;
v_opt(iV('x',':','e_y'))=Y_recov';
v_opt(iV('x',':','e_psi'))=wrapToPi(PSI_recov)';
writeXMLfile(v_opt,iV,spaceX,route)
```